

USENIX

DISTRIBUTED & MULTIPROCESSOR  
SYSTEMS (SEDMS II)

USENIX  
USENIX  
USENIX  
USENIX  
USENIX

## SYMPOSIUM PROCEEDINGS

Symposium on Experiences with  
Distributed and Multiprocessor Systems  
(SEDMS II)

March 21 - 22, 1991

Atlanta, Georgia

For additional copies of these proceedings write

USENIX Association  
2560 Ninth Street, Suite 215  
Berkeley, CA 94710 USA

The price is \$30 for members and \$36 for non-members.

Outside the U.S.A and Canada, please add  
\$20 per copy for postage (via air printed matter).

Past USENIX Distributed & Multiprocessor Systems Proceedings

Distributed & Multiprocessor Systems Workshop 1989 Florida \$30

Outside the U.S.A. and Canada, please add  
\$20 per copy for postage (via air printed matter).

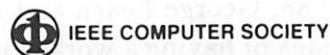
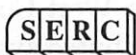
Copyright © 1991 by The USENIX Association  
All rights reserved.

This volume is published as a collective work.

Rights to individual papers remain  
with the author or the author's employer.

UNIX is a registered trademark of UNIX System Laboratories, Inc.  
Other trademarks are noted in the text.





# SEDMS II

## Symposium on Experiences with Distributed and Multiprocessor Systems

*Sponsored by*

**The USENIX Association**

**and**

**The Software Engineering Research Center**

*In cooperation with:*

**ACM Special Interest Group on Data Communication (SIGCOMM)**

**ACM Special Interest Group on Operating Systems (SIGOPS)**

**IEEE-CS Technical Committee on Distributed Processing (TCDP)**

**IEEE-CS Technical Committee on Operating Systems (TCOS)**

**IEEE-CS Technical Committee on Software Engineering (TCSE)**

*March 21-22, 1991  
Atlanta, GA*

## Introduction

In 1988, George Leach and members of the Florida West Coast Unix community got the idea of having a workshop or conference on distributed systems in the area. George contacted Peter Salus (then Executive Director of Usenix) and Gene Spafford about the idea, and plans were made. The decision was to have an event that would stress experiences with distributed and multiprocessor systems, rather than the theory and algorithms that seem to predominate at so many other conferences in the area. A formal proposal was made to the Usenix board, and they approved the workshop. Gene got the directors of the SERC (Software Engineering Research Center) to cover some of the expenses and clerical efforts, and to provide other assistance in publicity.

Thus, in the autumn of 1989, Usenix and the SERC cosponsored the first Experiences with Distributed and Multiprocessor Systems event. That was intended to be a workshop, but because of the quantity and quality of submissions and participation, it became a miniconference. It was known by the acronym WEBDMS, and included 25 presented papers, some of which were later developed into a special issue of the journal *Computing Systems*. The workshop was attended by over 125 people in Ft. Lauderdale, Florida, and was judged a great success.

We decided to see if the theme for this event could support a second event, this time as a more formal symposium requiring somewhat more developed papers. Thus, with Usenix and SERC as cosponsors again, and again with cooperation of the ACM and the IEEE Computer Society, we issued calls for submissions. The call was not widely publicized in academic journals, but we covered some large mailing lists and Usenet newsgroups.

The response was both gratifying and surprising. With only three months lead and limited publicity, we received 60 papers from researchers on 5 continents. The submissions dealt with everything from cache design to optical computing to multimedia workstations to performance tuning and debugging of multiprocessor systems. The program committee reviewed the papers, made some tough choices, and the 21 papers in this proceedings are the result.

We have already started to plan a third event, another symposium. It has already been approved by Usenix. It will be held in the spring of 1992, probably in the Western United States somewhere. Again, we expect Usenix and SERC cosponsorship, and ACM and IEEE-CS participation. We also hope you will consider contributing to that event, and in future SEDMS that others may put together in the years to come.

In the meantime, our thanks to the hardworking members of our program committee, and to all the reviewers who aided them in their reading and decision-making. We greatly appreciate the hard work, advice, and efforts on our behalf by Ellie Young, Carolyn Carr, and Judy DesHarnais of Usenix. Georgia Conarroe of Purdue proved herself invaluable (again) in helping keep Spaf on track with the program committee tasks. And thanks especially to all the people who took the time and effort to contribute papers to the symposium, and to come to Atlanta in March to be with us. Thank you — we hope you enjoy it!

George Leach  
General Chair

Gene Spafford  
Program Chair



# Program and Table of Contents

## Wednesday, March 20

*Registration and Reception*

7:00-9:00 pm

## Thursday, March 21

*Opening Remarks*

8:30

George Leach, General Chair (ATT Paradyne, Largo, FL)  
Eugene Spafford, Program Chair (Software Engineering  
Research Center/Department of Computer Sciences, Purdue  
University)

*Keynote Presentation*

8:45

How to Move Parallel Processing into the Mainstream?  
Professor H. T. Kung (School of Computer Science,  
Carnegie-Mellon University)

*Break*

9:45

*Session I System-Building & Experience*

10:15

Experience Developing the RP3 Operating System ..... 1  
Ray Bryant, Hung-Yang Chang and Bryan Rosenberg (IBM  
Research Division, Thomas J. Watson Research Center)

Experiences with Distributed Data Management in Real-time  
C3 Systems ..... 19  
Paul J. Fortier (Naval Underwater Systems Center), David  
V. Pitts, John C. Sieg, Jr. and C. Thomas Wilkes (Department  
of Computer Science, University of Lowell)

The ION Data Engine ..... 35  
Marc F. Pucci (Bellcore)

Building a Semi-Loosely Coupled Multiprocessor System  
Based on Network Process Extension ..... 51  
Helen S. Raizen (Prime Computer Inc.) and Stephen C.  
Schwarm (Digital Equipment Corporation)

*Lunch*

12:15 pm

*Session II RPC and Communications*

1:30 pm

Implementation and Performance of a Communication Facility for the RAID Distributed Transaction Processing System .... 69  
Enrique Mafla and Bharat Bhargava (Department of Computer Sciences, Purdue University)

Experience with Threads and RPC in Mach ..... 87  
Dan Duchamp (Computer Science Department, Columbia University)

Kernel-Kernel Communication in a Shared-Memory Multiprocessor..... 105  
Eliseu M. Chaves, Jr. (Universidade Federal do Rio de Janeiro, Brazil), Thomas J. LeBlanc, Brian D. Marsh and Michael L. Scott (Computer Science Department, University of Rochester)

*Break*

3:00

*Session III Scheduling / Synchronization*

3:45

Process Scheduling and Synchronization in the Renaissance Object-Oriented Multiprocessor Operating System..... 117  
Vincent F. Russo (Department of Computer Sciences, Purdue University)

A Hybrid Approach to Load Balancing in Distributed Systems ..... 133  
Prabha Gopinath (Philips Laboratories), and Rajiv Gupta (Department of Computer Science, University of Pittsburgh)

FALCON: A Distributed Scheduler for MIMD Architectures .... 149  
Andrew S. Grimshaw (Department of Computer Science, University of Virginia) and Virgilio E. Vivas (Department of Informatics, LAGOVEN, S.A., Venezuela)

*Short break*

5:15

*Work-in-Progress Session*

5:30

Moderator: Mike O'Dell (Bellcore)

*Reception & light buffet* 6:30-8:30

*Optional discussion/panel session* 8:30

Orthogonal Optimization Languages in Distributed Systems  
plus other discussion, debates, etc.

## Friday, March 22

*Planning for SEDMS III* 8:30am  
Leach/Spafford

*Session IV Debugging and Analysis* 8:45

Performance Evaluation of the Sylvan Multiprocessor Architecture ..... 165

Forbes J. Burkowski, Charles L. A. Clarke, Gordon J. Vreugdenhil (Department of Computer Science, University of Waterloo) and Crispin Cowan (Computer Sciences Department, University of Western Ontario)

Debugging Multiprocessor Operating System Kernels ..... 185  
Noemi Paciorek, Susan LoVerso and Alan Langerman (Encore Computer Corporation)

Debugging the Time Warp Operating System and Its Application Programs ..... 203  
Peter L. Reiher (Jet Propulsion Laboratory), Steven Bellenot (The Florida State University), and David Jefferson (UCLA)

*Break* 10:15

*Session V Performance Tuning* 11:00

Lock Granularity Tuning Mechanisms in SVR4/MP ..... 221  
Mark D. Campbell, Russ Holt and John Slice (NCR Corporation, E&M Columbia)

Measured Performance of Caching in the Sprite Network File System ..... 229  
Brent Welch (Xerox-PARC CSL)

*Lunch* 12:00

*Session VI Distributing Memory and Data* 1:30pm



Using Kernel-Level Support for Distributed Shared Data .....	247
David L. Cohn, Paul M. Greenawalt, Michael R. Casey and Matthew P. Stevenson (Department of Computer Science and Engineering, University of Notre Dame)	

Virtual Memory Xinu .....	261
Douglas Comer and James Griffioen (Department of Computer Sciences, Purdue University)	

Early Experience with Building and Using the Gothic Distributed Operating System .....	271
Isabelle Puaut, Michel Banatre and Jean-Paul Routeau (IRISA—INRIA, France)	

*Break* 3:00

*Session VII Distributed Systems* 3:30

Supporting an Object-Oriented Distributed System: Experience with UNIX, Mach and Chorus .....	283
F. Boyer, J. Cayuela, P. Y. Chevalier, A. Freyssinet, and Daniel Hagimont (Unite Mixte Bull-IMAG/Systemes, Gieres, France)	

Can We Study Design Issues of Distributed Operating Systems in a Generalized Way? .....	301
G. W. Gerrity, A. Goscinski, J. Indulska, W. Toomey and W. Zhu (Department of Computer Science, University College, University of New South Wales)	

Language and Operating System Support for Distributed Programming in Clouds .....	321
Partha Dasgupta (Department of Computer Science and Engineering, Arizona State University), R. Ananthanarayanan, Sathis Menon, Ajay Mohindra, Mark Pearson, Raymond Chen and Christopher Wilkenloh (Distributed Systems Laboratory, College of Computing, Georgia Institute of Technology)	

**Adjourn**

**5:00**

**Alternate Paper**

*Not scheduled for presentation:*

**Experiences with the Liason Network Multimedia Workstation**

**Howard P. Katseff, Robert D. Gaglianella, Thomas B. London, Bethany S. Robinson and Donald B. Swicker (AT&T Bell Laboratories)**

## The Program Committee

John R. Barr, Ph.D.  
Software Systems Research Laboratory  
Motorola, Inc.

Professor Bharat Bhargava  
Department of Computer Sciences  
Purdue University

Professor David L. Cohn  
Computer Science and Engineering  
University of Notre Dame

George W. Leach  
AT&T Paradyne

Michael D. O'Dell  
Bellcore

Professor Karsten Schwan  
College of Computing  
Georgia Institute of Technology

Professor Michael L. Scott  
Computer Science Department  
University of Rochester

Roger K. Shultz  
Collins Commercial Avionics  
Rockwell International

Professor Eugene H. Spafford, Chair  
Software Engineering Research Center  
Department of Computer Sciences  
Purdue University

Professor Satish K. Tripathi  
Department of Computer Science  
University of Maryland at College Park

## Other Reviewers (thanks!)

Ricardo G. Bianchini, William J. Bolosky, Michael R. Casey, Troy Cauble, Steve Chapin, Catherine F. Chronaki, Dennis J. Ciplickas, Gib Copeland, Prakash Ch. Das, William E. Garrett, Frank Greco, Paul M. Greenawalt, James Griffioen, Yenjo Han, Abdelsalam Helal, Karl F. Heubaum, Leonidas I. Kontothanassis, Edward W. Krauser, Dinesh C. Kulkarni, Evangelos Markatos, John O. Moody, Hsin Pan, Cesar A. Quiroz, James R. Robbins, Vincent F. Russo, John E. Saldanha, Lih-Chyun Shu, Neil G. Smithline, Matthew P. Stevenson, Colleen L. Templin, John M. Tracey, Karen M. Tracey, Jack E. Veenstra, Robert W. Wisniewski





# Experience Developing the RP3 Operating System<sup>1</sup>

Ray Bryant  
Hung-Yang Chang  
Bryan Rosenberg

IBM Research Division  
Thomas J. Watson Research Center  
P. O. Box 704  
Yorktown Heights, NY 10598  
{raybry,hychang,rosnrbg} @ ibm.com

## Abstract

RP3, the Research Parallel Processing Prototype, is a research vehicle for exploring the hardware and software aspects of highly parallel computation. RP3 is a shared-memory machine that was designed to be scalable to 512 processors; a 64 processor machine has been in operation since October 1988. The operating system for RP3 is a version of the Mach system from Carnegie Mellon University. This paper discusses what we have learned about developing operating systems for shared-memory parallel machines such as RP3 and includes recommendations on how we feel such systems should and should not be structured. We also evaluate the architectural features of RP3 from the viewpoint of our use of the machine. Finally, we include some recommendations for others who endeavor to build similar prototype or product machines.

## Introduction

The RP3 project of the IBM Research Division had as its goal the development of a research vehicle for exploring all aspects of highly parallel computation. RP3 is a shared-memory machine designed to be scalable to 512 processors; a 64-way machine was built and has been in operation since October of 1988.

For the past few years, the authors of this paper have been responsible for creating the operating system environment used to run programs on RP3. (The operating system for RP3 is a version of Mach [1] which is a restructured version of BSD 4.3 Unix.<sup>2</sup>) The extensions we made to Mach to support RP3 are described in [6] and will not be discussed in detail here. Instead, this paper summarizes our experience developing Mach/RP3 and presents our views on how operating systems for highly-parallel shared-memory machines such as RP3 should be constructed, as well as our experience in supporting and using this system for parallel processing research.

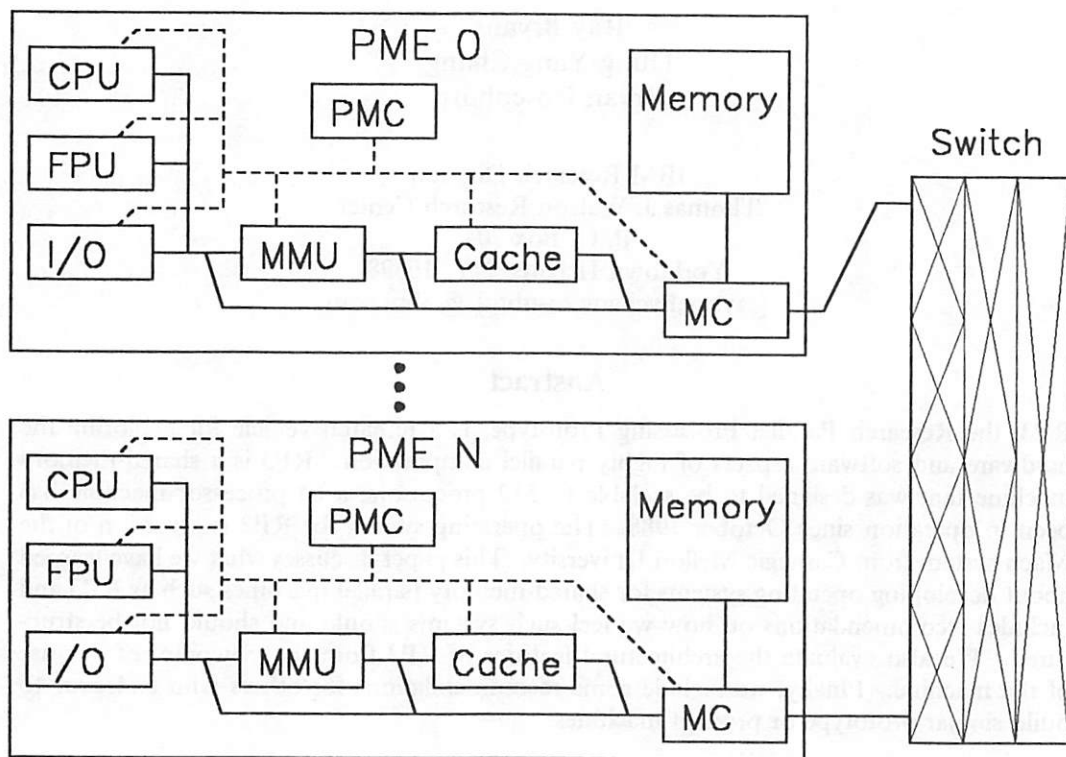
In the following sections of this paper, we provide an overview of the RP3 architecture and a brief history of the RP3 project. We then discuss the lessons we feel we learned during the course of this project and we state some recommendations we would make to developers of similar machines, whether such machines are designed as research prototypes or commercial products.

## RP3 Hardware Overview

Figure 1 illustrates the RP3 architecture. An RP3 machine can consist of up to 512 *processor memory elements* or PME's. The prototype hardware that was actually built, which we will refer to as RP3x, consists of 64 PME's. Each PME includes the following components:

<sup>1</sup> Supported in part by the Defense Advanced Research Projects Agency under Contract Number N00039-87-C-0122 (Multi-processor System Architecture).

<sup>2</sup> Unix is a registered trademark of AT&T in the United States and other countries.



**Figure 1. The RP3 Architecture**

- CPU** The central processing unit, a 32-bit RISC processor known as the ROMP. The same processor is used in the IBM RT workstation.
- FPU** The floating point unit, similar to the floating point unit found on the IBM RT System APC card. It uses the Motorola MC68881 floating point chip which implements the IEEE floating point standard.
- I/O** The I/O interface, which provides a connection to an IBM PC/AT that serves as an I/O and Support Processor, or ISP. Each ISP is connected to 8 PME's and to an IBM System/370 mainframe.
- MMU** The memory management unit. The MMU supports a typical segment and page table address translation mechanism and includes a 64-entry, two-way set associative translation lookaside buffer (TLB).
- Cache** A 32-kilobyte, two-way set associative, real-address cache. To allow cache lookup to proceed simultaneously with virtual address translation, the RP3 page size is made equal to the cache set size of 16 kilobytes.
- MC** The memory controller. The memory controller examines each memory request to determine whether it is for this PME (in which case it is passed to the memory module) or a remote PME (in which case it is passed to the switch). The top 9 bits of the address specify the target PME.



**Memory** A 1- to 8-megabyte memory module. (The 64-way RP3x is fully populated with 8-megabyte memory modules.) Note that all memory in RP3 is packaged with the processors.

**PMC** The performance measurement chip. This device includes registers that count such things as instruction completions, cache hits and misses, local and remote memory references, and TLB misses. It can also periodically sample the switch response time.

All the PME's of an RP3 machine are connected by a multistage interconnection network or switch. The switch, which is constructed of water-cooled bipolar technology, has 64-bit data paths and a bandwidth of roughly 14 megabytes/second per PME. All memory on RP3 is local to individual PME's but is accessible from any processor in the machine. However, a performance penalty is incurred when accessing remote memory. RP3x has an access time ratio of 1:12:20 between cache, local, and remote memory, assuming no network or memory contention. The fact that not all memory in the system has the same access times puts RP3 in the class of *non-uniform memory access* or *NUMA* machines. Support of this NUMA architecture required operating system extensions that are discussed in [6].

To avoid potential memory bottlenecks, the RP3 architecture includes the concept of *interleaved* memory. Addresses for interleaved memory undergo an additional transformation after virtual to real address translation. The interleaving transformation exchanges bits in the low- and high-order portions of the real address (see figure 2). Since the high-order bits of the address specify the PME number, the effect of the interleaving transformation is to spread interleaved pages across memory modules in the system, with adjacent double-words being stored in different memory modules. The number of bits interchanged (and hence the log base 2 of the number of modules used) is specified by the interleave amount in the page table. Figure 2 shows how the interleaving transformation can be used to spread virtual pages across multiple PME's.

Normally, all data used by more than one processor is stored in interleaved memory. For this reason, interleaved memory is also referred to as *global* memory. Local, or non-interleaved, memory is referred to as *sequential* memory.

If enabled in the hardware, a one-to-one hashing transformation is applied before the interleaving transformation. The hashing transformation randomizes sequential memory references as an additional technique to minimize memory conflicts.

The RP3 hardware does not provide any mechanism for keeping caches coherent between PME's; cache coherency must be maintained in software. The cache is visible to application code in the sense that instructions to invalidate the cache are provided in user mode. In addition, the segment and page tables include cacheability information, so that ranges of virtual addresses (on page boundaries) can be specified as cacheable or non-cacheable memory. Since there is no page table associated with real-mode memory access, all real-mode memory accesses on RP3 are non-cacheable references. Cacheable memory can be further identified as *marked data*. A single cache operation can be used to invalidate all data in the cache that has been loaded from virtual memory identified as marked data.

RP3 supports the *fetch-and-add* [9] operation (as well as *fetch-and-or*, *fetch-and-and*, etc.) as the basic synchronization primitive. *Fetch-and-add(location,value)* is an atomic operation that returns the contents of 'location' and then increments the contents of the location by 'value'.

Further details of the design of the RP3 PME and system organization can be found in [15] and [5]. RP3x, our working 64-way prototype, differs from the published design in the following respects:

- The I/O and Support Processors, or ISP's, in RP3x are simple IBM PC/AT's rather than the elaborate custom-built machines described in the published RP3 design. Each PC/AT is connected to 8 PME's and can access RP3 memory through its PME's. Memory requests from an ISP are identical to requests from a PME's own processor, so an ISP can address real or virtual memory that is local, remote, or even interleaved. The ISP-PME bandwidth is

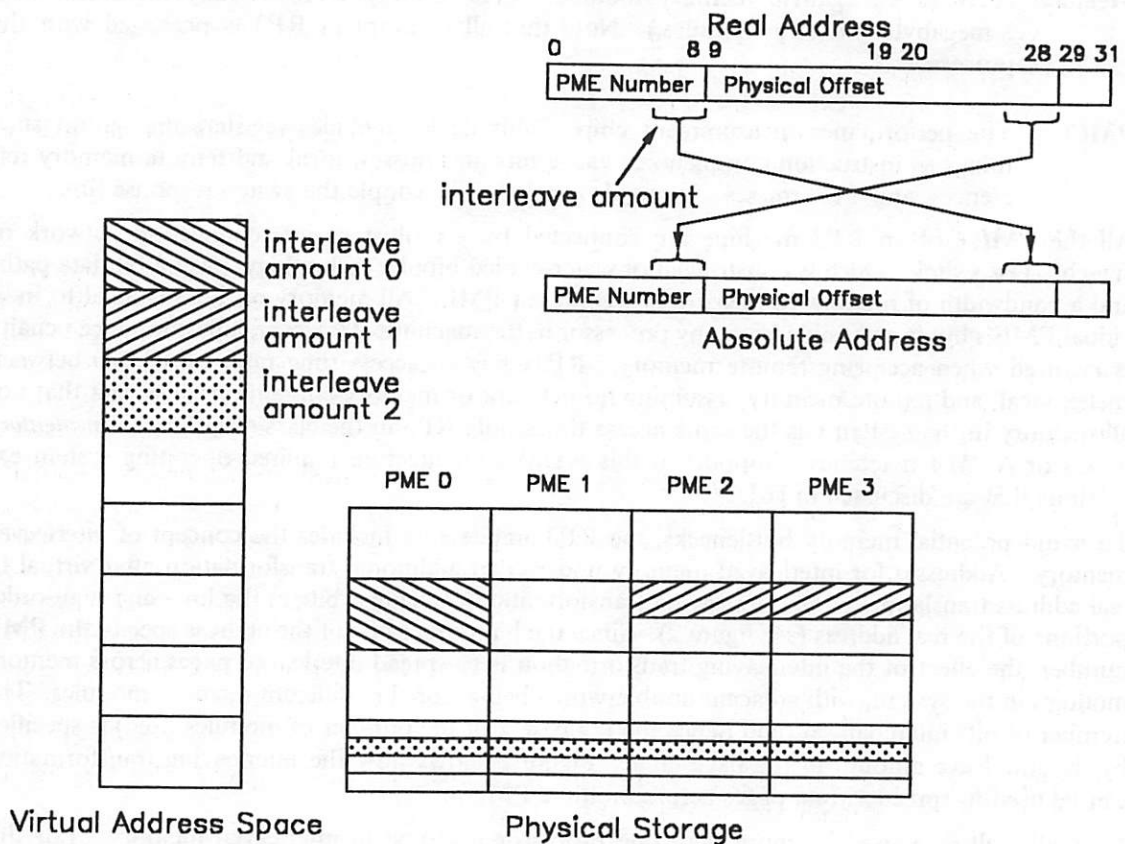


Figure 2. The RP3 interleaving transformation

roughly 500 kilobytes/second and can be dedicated to a single PME or multiplexed among PME's. An ISP can raise an interrupt in any of its PME's, and a PME can signal its ISP. The I/O hardware allows such a signal to interrupt the ISP, but our current ISP software is synchronous and periodically polls each PME's signal line.

In the original RP3 design, each ISP was to be directly connected to devices such as disks and networks. In the implemented design, the ISP's are channel-connected to a System/370 mainframe which in turn can access a large collection of disks and other devices. Bandwidth between an ISP and the System/370 is roughly 3 megabytes/second. RP3 I/O requests are passed from a PME to its ISP to the System/370 and back.

- The original RP3 design called for a *combining switch* that would reduce memory and switch contention by merging fetch-and-op operations when they collide at interior switch elements. The design could not be implemented in the technology available at the time. The current design supports the full range of fetch-and-op's, but the operations are serialized at individual memory controllers and are not combined in the switch.
- The floating point processors in RP3x are based on the standard RT workstation floating point unit and incorporate Motorola MC68881 floating point chips implementing the IEEE floating point standard. The original RP3 design called for vector floating point processors implementing the System/370 floating point specification.
- The RP3x cache system limits the PME clock rate to 7 MHz, about a third of the originally projected rate. Furthermore, the current memory controller is able to support just one out-

standing request to the memory subsystem at a time rather than the eight outstanding requests it was designed to handle.

- The RP3x memory management unit does not support hardware reload of the translation lookaside buffer (TLB). When a processor makes a memory request to a virtual address that is not mapped by the TLB, an exception is raised, and a software exception handler must explicitly load translation information for the faulting address into the TLB.

### ***History of RP3***

Our experience with RP3 is closely related to its development history, so it is useful to summarize the major milestones of this project. The original idea that the IBM Research Division should attempt to build a large parallel processor apparently originated with a task force led by George Almasi during the winter of 1982-83. The earliest the name *RP3* was actually used appears to be in the fall of 1983, when a group led by Greg Pfister began to design the RP3 architecture. In October of 1984, the IBM *Corporate Management Committee* agreed to fund the RP3 project.

With funding secured, the existing project team was expanded to include a design automation group, a processor design group, a technology group (responsible for constructing the machine and coordinating production of parts with the IBM development divisions), and a software development group. The software group initially concentrated on the development of parallel applications. Since no parallel hardware was available, the approach selected was to emulate a virtual parallel processor under VM/370. This led to the development of the EPEX [7] system and a significant library of applications were written using the EPEX parallel programming extensions to Fortran.

Other significant technical milestones:

<b>Dec 1984</b>	RP3 architecture frozen. With the exceptions noted previously, this is a description of the machine as it exists today.
<b>Aug 1985</b>	A set of papers on RP3 were published in the <i>Proceedings of the 1985 International Conference on Parallel Processing</i> . [15][5][16]
<b>Dec 1985</b>	Power/mechanical frame completed and installed in lab.
<b>Jun 1986</b>	Uniprocessor version of RP3 instruction level simulator completed.
<b>Aug 1986</b>	First version of Mach on RP3 simulator completed.
<b>Dec 1986</b>	First complete processor chip set assembled and tested.
<b>Apr 1987</b>	Final-pass chip designs released to manufacturing.
<b>Sep 1987</b>	EPEX environment ported to Mach/RT.
<b>Sep 1987</b>	First full PME with final-pass chips completed.
<b>Sep 1987</b>	Multiprocessor version of RP3 instruction level simulator completed.
<b>Oct 1987</b>	Mach/RP3 runs on first PME.
<b>Nov 1987</b>	Mach/RP3 runs under multiprocessor RP3 simulator.
<b>Feb 1988</b>	Mach/RP3 runs on two-processor hardware.
<b>Jun 1988</b>	Mach and three EPEX test applications run on 4-processor hardware.

Aug 1988	Mach and three EPEX test applications run on 8-processor hardware.
Oct 1988	64-processor prototype (RP3x) completed and turned over to software team.
Nov 1988	64-way application speedup experiments completed on three EPEX test programs.
Feb 1989	Mach/RP3 with cacheability control and interleave support completed.
Mar 1989	Mach/RP3 with processor allocation primitives and local memory support completed.
Jun 1989	RP3x upgraded to include cache and PMC.
Jul 1989	RP3x available to outside users via NSF net.
Mar 1990	RP3x upgraded with floating point coprocessors. (Before this, all floating point had been emulated in software.)

A final historical note: all the authors of this paper joined the RP3 project after the initial design for the machine was complete. Thus, we are unable to comment on some of the early RP3 architectural and design decisions.

### *Lessons Learned from RP3 Operating Systems Development*

**Mach was a win.** The original plans for RP3 included a contract with the Ultracomputer project at New York University for the development of a Unix-compatible RP3 operating system based on the Ultracomputer Symunix operating system [8]. For a variety of reasons, our group chose to pursue Mach, first as an alternative, and then as the primary operating system for RP3. The selection of Mach as the basis for the RP3 operating system was a successful strategy for the following reasons:

- It allowed us to use the same operating system on RT workstations, on VM/370, and on RP3. These versions of Mach cooperate in supporting compilation, debugging, and testing of user code on RP3. The same programming environments and compilers execute under Mach/RT as under Mach/RP3, and users are able to accomplish much of their debugging on Mach/RT before moving to the parallel machine.
- It enabled the rapid development of an initial uniprocessor RP3 operating system. Mach was designed to be portable and maintains a fairly clear separation of machine-independent from machine-dependent code. Since RP3 uses the same processor as the RT workstation, porting Mach/RT to a single RP3 PME was straightforward. Uniprocessor Mach/RP3 uses not only the machine-independent code from Mach/RT but much of the machine-dependent code as well. The major exception concerns memory management, because the RP3 and RT memory management units are radically different. Here again the porting effort was aided by Mach's clear encapsulation of machine-dependent memory management code.
- It aided the transformation of the uniprocessor RP3 operating system into a multiprocessor operating system. The machine-independent Mach code was multiprocessor-capable to begin with. We could concentrate on making the RT-based machine-dependent code multiprocessor-capable as well. In this effort we were aided by the examples provided by existing Mach implementations for a variety of commercial multiprocessors.
- It simplified the support of the RP3 memory architecture. Changes for global and local memory support as well as for user-level cacheability control were isolated in the machine-dependent portion of the kernel.

Some disadvantages of using Mach were that



- Although the Mach kernel executes correctly on shared-memory multiprocessors, in the version of Mach we used, most of the BSD Unix functionality is globally serialized. Furthermore, because it was designed for more or less generic multiprocessors, the Mach kernel does not make significant use of the sophisticated RP3 fetch-and-op synchronization primitives. The NYU Symunix system was designed specifically to avoid these limitations, but in our experience the problems have not been severe, a subject on which we will have more to say in a later section.
- Mach was not designed for NUMA multiprocessors. Adding support for the RP3 NUMA architecture has been a major effort, albeit an effort that was aided by the modular design of the Mach memory management system.
- Given the final speed of the PME's on RP3, and the single user mode in which it is normally used, the Mach/RP3 kernel is more system than is actually required on RP3. We don't run RP3x as a time-sharing Unix system, so we don't really need to run a full-function Unix system. It may have been more appropriate to use a small run-time executive that implements the small set of system calls that our applications actually use. The difficulty with this approach is choosing an appropriate set of system calls. New calls may be required as more and more applications are ported to the machine. Furthermore, for operating system researchers, Mach is more interesting than a small run-time executive, and this factor played a key role in our choosing Mach.
- Mach is a large system. Changes to the system often required inspection of large portions of code that were irrelevant to the problem at hand. A small kernel might have been easier to deal with.
- Mach IPC on the Mach 2.0 kernel we are running is too expensive. Mach IPC is used not only for communication between user tasks, but also for communication between a user task and the kernel, at least for Mach-specific kernel functions. IPC performance may have been improved in Mach 2.5 and subsequent releases, but for our kernel a Mach system call is significantly more expensive than a BSD system call, and the difference is largely due to IPC overhead.

**Functional simulation was extremely important.** Early in the project we obtained an instruction-level simulator of the RT workstation, and we were able to convert it first into a uniprocessor RP3 simulator and then into a multiprocessor simulator. The time invested in this effort was considerable, but it was time well spent because it let us develop system software well ahead of the hardware development schedule. The first version of Mach/RP3 was available under the RP3 simulator more than a year before the first prototype PME was completed. The Mach/RP3 kernel came to be regarded as the final architectural verification test for each new version of the prototype hardware. Without the simulator, we would never have had the confidence in the correctness of Mach/RP3 to use the kernel for this purpose.

Without the RP3 simulator, we would probably not have completed the Mach/RP3 kernel until months after RP3x was built. The RP3 PME development plan required the custom design of several chips. Chip design "front loads" the processor development cycle in the sense that for the first two-thirds of the development cycle no prototype hardware is available. Once a set of correct chips has been completed, production of the parallel machine occurs at an accelerated rate. For the RP3 project, the first functioning PME was available two years after the project started; RP3x was completed approximately one year later. By using the RP3 Functional Simulator, we were able to verify not only that the kernel was correct for the target architecture, but that the applications would execute correctly as well. We were ready to execute kernel and applications code on the 4-, 8-, and 64-way machines as soon as they were available. Without this ability, we would have fallen behind the accelerated progress that occurred in the hardware side of the prototyping effort during the last year of the hardware development cycle.

Even if the PME design cycle had been much shorter, the simulator would have been valuable because it provides a much more productive development environment than any prototype hardware.



We found the conversion of a uniprocessor ROMP simulator to a simulator of RP3 to be a detailed, but relatively straightforward process. Aside from architectural changes between the ROMP and the RP3 PME, the hardest part of the problem was simulating the multiprocessor. Rather than rewrite the simulator, we converted it into a parallel program by replicating the simulator code in multiple virtual machines under VM/370, and using shared memory between the virtual machines to represent RP3 memory. Since the simulator itself was run on a multiprocessor System/370, we were able to find and eliminate many multiprocessor timing bugs before the multiprocessor RP3 hardware was available. If we had run the simulator on a uniprocessor system, interactions between simulated processors would have been limited by VM/370 dispatching intervals, and we probably would not have found as many timing bugs.

**It was not necessary to significantly restructure the Mach kernel to achieve significant application-level parallelism for computation intensive workloads.** First of all, let us point out that Mach/RP3 is a version of Mach 2.0, and does not include the changes for multiprocessor Unix completed by Encore [4]; these changes were subsequently picked up by the Open Software Foundation and are a part of the OSF/1 kernel. Thus, while the Mach portion of the kernel we are running is parallel and symmetric, the Unix portion of the code essentially runs under a single lock. That is, the Unix portion of the kernel is executed exclusively by a single processor called the *Unix Master*. *Slave* processors run user programs, Mach code, and trivial Unix system calls. All other Unix system calls are implemented by suspending the calling thread on the slave processor and rescheduling the thread to run on the master processor.

In spite of this (relatively severe) bottleneck in this version of the Mach kernel, we have been able to achieve 40-way speedups on the 64-way RP3x system with relative ease. We attribute this to the fact that our workloads are typical engineering-scientific programs. A typical program issues a number of system calls to create separate processes and establish a common shared-memory region, but once it begins computation, it issues relatively few system calls. (To some extent the workload is artificial, since users know that RP3 has limited I/O bandwidth and hence do not run I/O intensive jobs on the machine. Nonetheless, we feel it is a workload characteristic that such jobs issue far fewer system calls per million instructions than a commercial workload might.)

Originally, it was felt that more system restructuring would be necessary for RP3. For example, it was a commonly held opinion that we would have to modify the system dispatcher to use highly-parallel, non-blocking queue insertion and deletion routines based on *fetch-and-add* [9]. However, we have never found the dispatcher on RP3 to be a significant bottleneck, in particular because our philosophy is to allocate processors to user tasks and to let users do local scheduling. The system dispatcher is only used for idle processors and global threads that are not bound to particular processors. The scheduling problem thus divides naturally into two levels: system-level scheduling decisions that are made on a job by job basis, and user-level decisions that are made on a thread by thread basis. (A two-level scheduling mechanism of this flavor is described in [3]). The intervals between system-level scheduler events are on the order of many seconds to a few hours; user-level scheduling events can occur as frequently as once every few hundred instructions. Thus the system-level scheduler should not be a bottleneck and need not use *fetch-and-add* algorithms. The user-level scheduler is part of the application and if necessary can use a *fetch-and-add* queue in user space to reduce local scheduling overhead.

**A 64-processor machine is much different from a 4- or 8-processor machine.** A similar observation concerning the BBN Butterfly was made in [11]. We learned this lesson in October 1988 when we first tried to boot our kernel on the 64-processor machine. Before this, we had successfully run our kernel and a small set of applications on the 4-way and 8-way prototypes, and had booted the kernel under a 64-processor version of the RP3 Functional Simulator. On the 4-way and 8-way machines, it took only a few day's effort, once the hardware was available, to get our kernel and application suite running. (This success, of course, depended on much previous work with the simulator and with one- and two-way versions of the hardware.) We did not encounter new timing bugs in the kernel when moving from the 2-processor to the 4- or 8-processor systems, and reasonable kernel startup times and application speedups were easily achieved. However, when we attempted to bring up the 64-way kernel, we found to our surprise that:

- New timing bugs appeared (the kernel would not boot reliably).
- Kernel startup time had expanded to an unacceptable 2.5 hours!

Of course, we were aware that the kernel we had available at that time was far from optimal (it did not use interleaved memory, for example) but we were surprised nonetheless by the disparity between the kernel startup times for the 8-way and 64-way machines. Eventually we were able to reduce the kernel startup time to around 8 minutes by reducing network contention due to spin locks, by placing the kernel in interleaved memory, and by exploiting the processor caches on RP3x.

**NUMA machines (like RP3) share the advantages and disadvantages of both tightly-coupled and distributed multiprocessors.** On a message-passing machine, a serial program must usually be significantly restructured before it can even be loaded onto the parallel machine. The primary advantage of the shared-memory parallel processing approach is that it is relatively easy to get an existing, serial program to work in the parallel environment, and on RP3 we have found this to be the case. Transformation of serial programs to the EPEX environment is relatively straightforward [7]; once an application has run under EPEX on the RT, it is a simple process to move it to RP3.

However, even though the program may run correctly on RP3 with relatively little restructuring, to achieve the maximum available speedup may require significant program restructuring, because multiprocessor speedup may be limited unless one uses such RP3 features as local and cacheable memory.

For programs in the EPEX style, code and private data are replicated in each address space and thus can be placed in local and cacheable memory by the language run-time. With this optimization speedups in the 40's can be attained on the 64-processor machine with relative ease. To improve the speedup, additional code restructuring is required.

For example, one of our colleagues (Doug Kimelman) was able to obtain a speedup of 57 using 64 processors for a parallel implementation of a prime number counting program based on the sieve technique. To do this he had to:

- divide the sieve space up into 64 subranges.
- place each sieve subrange in local memory on a processor.
- execute separate sieve algorithms on each processor.
- use fetch-and-add to accumulate the number of primes found in the subrange into the total number of primes found.

Exactly this kind of program repartitioning would also allow the sieve algorithm to work well on a message-passing machine.

The point is that in order to get good speedups on any kind of fine-grain, highly-parallel computation, one must concentrate on partitioning the data in such a way as to minimize data movement and interprocessor synchronization. This statement is true for either shared-memory or message-passing architectures. On RP3, it is our belief that while it is easy to convert serial code to parallel code and to get it to execute correctly, obtaining maximum speedups requires the same kind of restructuring that is required to get the program to execute on a message-passing machine. We still feel that the overall effort is smaller on RP3 than on a message-passing machine, because the existence of shared memory lets the programmer concentrate on restructuring only those parts of the program that are critical for good parallel performance. Distributed shared memory systems such as that of Kai Li [12] may alleviate this disadvantage of message-passing systems, but the amount of restructuring required to achieve a given level of performance will still be greater on such systems than on true shared-memory systems because the penalties for non-local memory access are so much greater.

**Interleaved memory lets us program the RP3 as a true shared-memory multiprocessor.** We are able to run RP3x using a single copy of the kernel text and data; partitioning of kernel data structures on a per-PME basis is not required. Furthermore, optimizations such as copy-on-write and shared

user text segments are feasible in interleaved memory. Use of these optimizations in sequential memory can result in intolerable memory contention.

Early versions of Mach/RP3 did not support interleaved memory. All kernel text and static data resided in a single memory module, and most application programs were small enough to fit in a few 16-kilobyte pages. For these versions of the kernel, we found memory and network contention to be significant, especially the contention caused by instruction fetches. (In particular, instruction fetches from the busy-wait loop in the kernel *simple\_lock* routine were one of the key reasons that kernel startup on the 64-processor prototype took so long.) Furthermore, text and read-only-data regions of related tasks are shared copy-on-write. This optimization can reduce memory usage and copying costs, but it can result in severe contention for the memory module holding the single shared copy. Application programmers found it necessary to artificially modify the read-only regions of their programs to force them to be replicated.

These problems were alleviated when we restructured the Mach/RP3 machine-dependent memory management system to support the RP3 interleaving mechanism. Without interleaving, we would have been forced to replicate the kernel text to every processor and to disable the Mach kernel optimizations that result in user-level memory sharing, and we would have had to devote a considerable effort to partitioning the kernel data structures into PME-specific regions.

Our early memory contention problems were exacerbated by the initial lack of processor caches in RP3x. Processor caches can alleviate some of the contention caused by instruction fetches and accesses to read-only or private data. Nevertheless, interleaving of non-cached data structures is still important, and even for cached pages, interleaving increases the bandwidth available for cache reload.

**Traditional ideas of processor allocation and system control do not necessarily apply in the shared-memory parallel-processing arena.** In [14] Pancake and Bergmark note the discrepancy between the approach to parallel programming taken by computer scientists and that taken by computational scientists. We encountered this distinction when we first started work on the RP3 operating system. We were somewhat shocked by the attitude of the (potential) RP3 user community toward operating systems and operating system scheduling. Instead of regarding the system as a convenient environment for parallel programs, some of our users regard the operating system as an adversary bent on denying them direct access to the hardware.

For example, in our view the operating system has the right to suspend any process at any time based on the operating system's concept of the importance of that process. Our users explained to us, patiently, repeatedly, determinedly, and when necessary, vehemently, that this approach wreaks havoc with parallel programming models that do their own processor allocation. For example, in the EPEX model of computation, the program determines the number of processors available to the job and divides FORTRAN *DO*-loops across the available processors. Each processor is assigned a subset of the *DO*-loop indices for execution. Subsequently, a *BARRIER* statement is used to ensure that all loop instances have completed. Since it is assumed that the computation has been divided equally among the processors, the *BARRIER* is implemented using spin locks rather than suspend locks. Each processor is represented in the EPEX program as a separate Unix process. If the operating system were to suspend one of these processes after the parallel *DO*-loop has started, the remaining processors loop endlessly when they reach the *BARRIER* statement waiting for the last processor to complete its part of the computation.

Similarly, a job may only be able to adapt to a change in the number of processors at certain points in its execution. EPEX programs cannot adjust to a change in the number of processors during execution of a parallel *DO*-loop. Only before the loop has started or after it has completed can the number of processors be changed. Even then, the allocation of additional processors requires the creation of additional Unix address spaces in the EPEX program. The only realistic alternative for the EPEX model appears to be to statically allocate processors to the program when it begins executing. This, of course, conflicts with the ability of the system to run other jobs, to service interrupts, or to multiprogram the system, none of which were of interest to our users.



Since a significant number of applications for RP3 had already been written for the EPEX model of computation, we could not afford to simply ignore their concerns. Instead, we developed the concept of *family scheduling* [6] which corresponds to the idea of *gang scheduling* or *co-scheduling* [13]. With the family scheduler extensions, the Mach/RP3 system will never suspend individual processes in a parallel program, but it can, if necessary, reclaim processors from a family by suspending the entire family.

We believe that the family scheduler is an acceptable compromise between our user's needs and the requirements of the operating system to perform global scheduling. The EPEX run-time library was enhanced to use the family scheduling primitives on RP3 and the PTRAN [2] compiler, an automatic parallelizing compiler for FORTRAN, also uses the family scheduling primitives. In the end, however, the facilities of the family scheduler have been largely underutilized, since most users prefer to run applications on RP3x in a single-user-at-a-time mode in order to obtain repeatable execution times.

**Repeatability of execution times has been a problem for us on RP3.** Our users are primarily interested in studying parallel algorithm speedup. This task is difficult if individual timing runs are not repeatable. Variations in execution time from one run to the next can be caused by a variety of reasons.

On RP3, multiple jobs executing simultaneously can interfere with each other even if there are enough processing resources to satisfy all of their requirements. Contention for memory bandwidth, for processing time on the Unix master processor, and for I/O bandwidth can degrade the performance of even well-behaved programs. Most jobs are therefore run on the machine in a single-user-at-a-time mode.

Even on an otherwise idle machine, execution times can vary from one run to the next. Of course, the variance can be due to gross non-determinism in a true parallel program, but even subtle non-determinism can affect performance. In successive runs a given processor might execute exactly the same sequence of operations but on different regions of a shared data structure. Even if the data structure is laid out uniformly in interleaved memory, operations against different regions may result in different remote access patterns, and consequently in different levels of network and memory contention.

Variations in the execution times of completely deterministic applications can be due to non-deterministic placement of kernel data structures associated with the application. For example, process context blocks and per-process kernel stack segments are dynamically allocated when an application starts. These structures might therefore reside at different kernel virtual addresses from one run to the next, and consequently might be spread across different sets of PME's. This problem is usually not severe for computation-intensive applications, although clock-interrupt overhead is occasionally increased because of a particularly unfortunate placement of a kernel data structure.

Our users have learned to make sure the same version of the operating system kernel is used for an entire sequence of timing runs. Trivial changes in the Mach/RP3 kernel can cause timing differences in user applications, because the kernel itself is located in interleaved memory. Small changes in the kernel can shift key data structures in such a way that imbalances in memory reference patterns can appear (or disappear). Unfortunately, even simple bug fixes can change the layout of memory. In one case, a 50-byte change in initialization code (only executed at boot time!) dramatically changed key performance measures of a particular user program.

A final point concerns the RP3 address hashing mechanism. With hashing disabled, the PME location of a particular double-word of an interleaved page is determined solely by the word's virtual address. With hashing enabled, the location is a function of both the virtual and real addresses of the double-word. Applications can explicitly control the virtual addresses they use, but they have no control over (or even knowledge of) the real addresses they use. In repeated runs of a program, real memory addresses of the user's virtual pages will change, and hence remote access patterns will change from one run to the next, even if the application makes deterministic use of its virtual address space.

**The RP3 variable interleave mechanism is too inflexible to be used easily in other than trivial ways.** The RP3 interleaving transformation can spread a virtual page across a set of PME's smaller than the entire machine, but it cannot spread a virtual page across an arbitrary subset of the machine. The interleave amount in the page table is the logarithm base 2 of the number of memory modules used, and the page is interleaved across a range of adjacent memory modules that begins at some multiple of this amount. For example, if the interleave amount is 4, the page is spread across 16 consecutive memory modules, starting with PME 0, 16, 32, or 48. These restrictions make it difficult to interleave pages across just those processors allocated to a particular parallel application. Furthermore, an interleaved page slices across the real memory of the machine and occupies part of a page frame in each memory module. The rest of the memory in the affected page frames can only be used in identically interleaved virtual pages. Supporting variable interleave amounts would have made the allocation of real memory a two-dimensional bin packing problem. Also, given the presence of processor caches, it seemed most appropriate to us to use the maximum interleave amount so as to make the maximum memory bandwidth available for cache reload. For our purposes, a single page table bit indicating whether a page is located in sequential memory or is interleaved across the entire machine would have been sufficient. An environment where the variable interleave amount would be useful is described in the next section.

**Multistage interconnection networks do not lend themselves to construction of traditional multi-user machines.** As mentioned above, we normally run RP3x in a single-user-at-a-time mode. Given our application set and user community we would suggest that other designers of machines with memory hierarchies implemented by multi-stage interconnection networks not attempt to run the system with a single kernel, but instead adopt a partitioning approach similar to that proposed for TRAC [17].

On RP3, one could use the variable interleave amount to dynamically partition an RP3 into sub-machines on power-of-two boundaries. To keep partitions from interfering with each other, each partition could be given its own copy of the kernel, with each page in a partition being either a sequential page or a page interleaved across the entire partition. The machine would be effectively split into distinct sub-machines as far as the switch and memory are concerned. Assuming that the I/O system were reconfigured to split along similar lines, this would give users the size machine they want, with strong guarantees of repeatable performance and non-interference from other users.

Such a dynamic partitioning would require a controlling system to run outside of RP3 itself. A natural place to run this system would be in the I/O and Support Processors or ISP's in the system since these machines already support system start-up. We have not pursued this approach because the software to do so would be complex and would have to run in the primitive environment of the current ISP's. However, this would probably be a better match to both our users' needs and the memory and processor structure of the machine. Additionally, it would improve processor utilization over our current method of operation, since a single-user job that uses only a few processors leaves most of the machine idle.

**The potential for a software-controlled cache to be as efficient as a hardware-controlled cache was not demonstrated by RP3.** No compiler that could exploit the RP3 software-controlled cache was ever completed. Without compiler support, cache invalidation and software cache coherency are difficult to implement efficiently. To ensure safety, hand-coded software cache coherency schemes are forced to invalidate the cache too frequently and processor performance is dominated by cache cold-start effects. The RP3 "marked data" mechanism helps alleviate this problem but does not eliminate it.

Experiments indicate that, even if RP3 had hardware cache-coherency, it would often be advantageous to keep shared variables in non-cacheable memory. Shared data is often read once and not reused quickly, and loading it into the cache may evict data that could profitably be retained. This effect is particularly severe on RP3 because instructions and data share a single cache.

The difficulty of dealing with RP3's non-coherent cache structure has led us to execute the Mach kernel code with instructions and stack in cacheable memory, and all other data in non-cacheable memory. The result is that kernel code executes half as fast as user code that places all data in

cacheable memory. A hardware-coherent cache would have allowed us to place all kernel data in cacheable memory. The performance penalty for supporting a hardware-coherent cache would probably have been no worse than the performance penalty we now incur due to data being non-cacheable. Furthermore, a hardware-coherent cache would allow the use of the cache when the processor is executing in real mode. On RP3, all real-mode accesses are non-cacheable. While very little code on Mach/RP3 executes in real mode, the performance penalty of executing in non-cacheable mode is high and is one of the reasons that we have encountered significant performance problems related to TLB thrashing (see below).

Our current belief is that the correct architecture would be for the system to support both coherent and non-coherent modes of execution. The overhead of hardware cache coherence protocols on highly-parallel machines may be unacceptably high for some applications. Code (such as the Unix kernel) that is too difficult to restructure to allow the use of software-level cache coherency would be executed in coherent mode. Restructured software that can gain from the reduced hardware overhead would execute in non-coherent mode.

**Hot spot contention, as defined in [16] is not a problem on RP3x.** Contrary to previous predictions of "hot-spots" on RP3, our measurements show insignificant delay due to hot-spot tree saturation on the 64-way prototype. In [18] Thomas reports that tree saturation is also not a problem on the BBN Butterfly, because the Butterfly switch is *non-blocking*. The RP3 switch is *blocking*, so for us the discrepancy between prediction and measurement has a different explanation. The original RP3 design, on which the tree saturation prediction was based, allowed up to 8 outstanding memory requests per processor. The RP3x prototype only allows one outstanding request per PME. This engineering change makes RP3x a processor- and memory-bound machine, not a communication-bandwidth-bound machine.<sup>3</sup> Indeed RP3x can be regarded as a 64-processor machine with a 128-port global memory. 64 of the ports are dedicated to the local processors, leaving 64 ports to satisfy global memory requests. In order to avoid saturating a memory module, the pattern of memory references thus has to be nearly uniform since there is no extra capacity available to deal with an imbalance. In hindsight, it would have been better to construct the machine with 256 or more ports to global memory so that when the memory reference pattern is not completely uniform, there is extra bandwidth available to service the requests at the more heavily-loaded memory modules.

**TLB thrashing can be a problem for systems using software TLB reload.** On RP3, the Translation Lookaside Buffer, or TLB, is a 2-way set associative cache of virtual to real address translations. As usual, this cache is used to avoid the penalty of inspecting the page table during each virtual address translation cycle. On the 64-way prototype, RP3x, TLB reload is performed in software at an expense of about 200 instructions per entry reloaded. Since these instructions must be executed in real mode, instructions and data are not cached and therefore a TLB reload requires approximately 2 ms. processing time. As a result, when TLB thrashing occurs, it has a significant effect on program performance. This problem came to our attention when a user informed us that a parallel loop of 8 instructions took more than 100 times as long to execute on one of the 64 processors in the system as on the others. Our initial guess was that the processor in question had a hardware problem, but subsequent runs showed the problem moved from processor to processor in the system. Through use of the PMC we were able to determine that the slow processor was taking an enormous number of TLB misses (and hence reloads). It happened that the loop was so constructed that one out of the 64 processors involved was trying to address local data, local instructions, and global data, all using the same TLB entry. Since the TLB is only 2-way set associative, each pass through the 8 instruction loop was causing three TLB misses, expanding the effective size of the loop from 8 to over 600 instructions. In general, we would recommend at least a 4-way set associative TLB be used if software TLB reload is proposed for a particular machine.

**Using existing software from a uniprocessor system is a win, but there are pitfalls.** The key advantage of using existing software is that it allows one to concentrate on the areas of the system that

<sup>3</sup> This change also eliminates the need for the RP3 *fence-registers*[5] and makes RP3x a serially consistent machine with respect to store order.



need to be modified. For example, since RP3 uses the same processor as the RT system, essentially all the utility commands on RP3 (*sh*, *ls*, *ed*, *fsck*, etc.) were taken unchanged from Mach/RT. We were also able to modify and use the kernel debugger from Mach/RT as the kernel debugger for RP3.

Since the machine-independent portions of the kernels for Mach/RP3 and Mach/RT are essentially identical, application programs compiled for the RT will execute on RP3x provided they do not use instructions supported only on the RT (such as *test-and-set*). Differences between the machines have been isolated to library routines. (For example, a subroutine on the RT simulates the *fetch-and-add* instruction with a variable protected by a *test-and-set* lock.) We have been able to test user code on Mach/RT and then move this code to RP3x for execution with relative ease.

Overall, we have been very successful in using Mach/RT tools for program development on RP3. The compilers we use on a routine basis are the same ones we use on Mach/RT. (These compilers, in turn come from IBM's AOS 4.3 for the RT system.) However, floating point support on RP3 has been complicated by the multiplicity of hardware floating point options available on the RT. Because of this, RT compilers do not generate native machine code for floating point hardware. Instead, they generate pseudo-code that is converted at execution time to match the type of hardware floating point card installed on the target machine. We go through this process during execution on RP3x, even though RP3x uses only the MC68881 coprocessor chip found on the RT APC.

The code translation is performed the first time a particular floating point operation is encountered during execution. Conversion involves replacing the pseudo-code sequence with compiled instructions appropriate for the floating point hardware. On RP3, the result is that we incur additional serialization overhead during this floating point code translation, since code shared by multiple processors must be converted only once. Occasionally, the compiler will not allocate enough memory to hold the translated code, and the floating point code generator will have to execute a *malloc* call to dynamically allocate memory to hold the generated instructions. The *malloc* call can result in a system call, seriously affecting application performance.

This type of floating point code generation does have its advantages, however. Floating point hardware for RP3x was not installed until June of 1990. Until then, we used software-emulated floating point. One of the options supported by the RT floating point code generator was "no floating point card available". In this case, the floating point pseudo-code generated by the compiler was not translated at all; instead its execution was emulated in software. This mode of execution was used as a method of supporting floating point operations on RP3x before the floating point coprocessors were installed.

One last suggestion we would make about exploiting existing software on a multiprocessor machine would be to make sure that the memory subsystem will support all commonly used instructions. In the original RP3 architecture, the ROMP *partial-word-store* instructions (*store-byte* and *store-half-word*) were not supported. If a processor encountered such an instruction, it would raise a program exception interrupt. The plan was to eliminate all such instructions from the kernel and application code by using a new compiler. This proposed compiler never materialized. Instead we had to resort to post-processing the assembly language output of our C-compiler. In this manner we avoided partial-word-stores in the kernel itself and in application code that could be recompiled. To avoid having to recompile all user commands, we wrote a program exception handler to emulate the partial-word-store instructions encountered in user mode.

This emulator had the following disadvantages:

- It could not be made to execute both efficiently and correctly when other processors were also using the emulator to update portions of the same target word. (Without setting a lock of some kind, the read-modify-write cycle of the emulation code could overwrite a byte modified by another processor.)
- It was extremely slow.

- It required emulation of not only the partial-word-store instructions themselves, but also of those *branch-with-execute* instructions whose execution target was a partial-word-store instruction.

By demonstrating the software overhead of partial-word-store emulation, we convinced the hardware designers to change the architecture to support these instructions..

**I/O architecture is hard; processor and system architecture is easy. Put more effort on the hard parts!** With RP3, as with many other designs, the fun part of developing the architecture was designing the virtual memory architecture, the system structure, the synchronization primitives and the like. Details related to I/O were regarded as being of secondary interest, since, after all, scientific programs were thought not to require significant I/O. The result is that RP3x has an "ad hoc" I/O architecture with a maximum bandwidth per ISP of about 500 kilobytes per second. This has limited our ability to experiment with performance visualization [10], to study seismic algorithms, and to do graphics on RP3, and has limited the speed of routine interaction with RP3x.

Creating a good I/O architecture requires real skill and detailed knowledge of the characteristics of I/O attachment. We have no simple solution to this problem other than to suggest that it is an important area that requires significant attention in the construction of any real parallel processor.

**The PMC has been tremendously useful.** The performance measurement chip on RP3 has been an excellent facility for understanding RP3 system performance. It was also a relatively simple chip to implement, since it is mostly made up of counters and interface logic that lets the processor read and reset the counters. The hard part of the design was in identifying and routing the signals from other parts of the PME to the PMC. Effectively what we have in the PMC is a small per-processor hardware monitor. During the past two years we have used the PMC to identify the TLB-thrashing problem discussed above, to measure the latency of the global memory in the presence of contention, to count local and global memory references, and to observe cache utilization. We have thus found the PMC to be crucial to the understanding of parallel system and application performance on RP3x. As processors are built of denser and denser integrated logic, it becomes more and more difficult to obtain these kinds of measurements by any other means. As integration densities continue to increase, it becomes more feasible to allocate a few circuits for performance measurement. We would therefore recommend that such instrumentation be included as a standard part of future processors.

**Don't overstate your goals.** The RP3 project has been criticized for not meeting the goals discussed in the papers published in the 1985 ICPP proceedings [15] [5]. In [15], the prediction was made that a 512-way RP3 would achieve a peak performance of 1.2 BIPS (assuming a 100% cache hit rate on instructions and data). Needless to say, we did not achieve that mark. But we did come within a factor of 3 of that prediction when it is scaled down to a 64-processor prototype: 1.2 BIPS per 512 processors is 2.3 MIPS per processor. In user mode, with data and instructions cacheable we routinely execute at 750 KIPS. The discrepancy is explained by the fact that we execute at one-third the clock rate originally proposed. This decrease in clock rate was made to circumvent timing errors in the final-pass network interface chips. Rather than wait for yet another pass of chips, the decision was made to push forward and complete a somewhat slower machine. This is a reasonable and defensible decision for a research project to make. Why is it then, that RP3 is regarded by many people as an uninteresting machine?

We feel this is due to several factors:

- When RP3x was first assembled in 1988 it included neither processor cache nor floating point hardware. When one takes a system designed to include cache and hardware floating point, and runs it without cache and with software-emulated floating point, performance is going to be disappointing. The final upgrade of the machine to include an RT-equivalent floating point unit was not completed until 1990.
- Since the NMOS ROMP in RP3x was announced by IBM in the original RT, we have seen a new version of the RT arrive (the APC, a CMOS ROMP) and it in turn has been replaced by a new generation RISC processor, the RISC System/6000. The floating point performance

of a single RISC System/6000 exceeds the performance of the 64-processor RP3x. But the point of RP3x was to do research into parallel processing, not to compete with the fastest processors available. RP3x has always been slower than a CRAY Y-MP, for example.

The real problem we feel is that the goals of the RP3 project were overstated. We have met the more realistic goal of creating a reliable research machine oriented to studying the hardware and software aspects of parallel, shared-memory computation. If anything, the RP3 project should be faulted for being too aggressive in terms of the multitude of architectural features it attempted to evaluate. We were unable to implement all of these facilities and some of the facilities that were implemented we have yet to evaluate carefully (hashing, for example). It probably would have been more reasonable to implement a smaller set of features well rather than so many features on a less than perfect machine.

**Flexible, but slow hardware is not as interesting to users as inflexible, but very fast hardware.** In the end, the success of RP3x is determined by its users. We wanted to build a machine that was flexible and sufficiently fast to attract application users to the machine. The hardware is very flexible, but unfortunately, it is not fast enough to attract applications. Instead, only the parallel processing expert is interested in using the machine, while applications users have moved on to more modern hardware. In general, this may be a generic problem for the parallel processing community because:

**It is hard to get ahead of the RISC microprocessor development curve using parallel processing.** Since we started the RP3 project in 1984, we have seen the development of office workstations with 15 times the processing power of the original RT. Advances in RISC processor technology appear to continue with processor speeds doubling every year or two. If in this environment a parallel processing machine is built based on a particular microprocessor, and if the lead time for this development effort exceeds a few years, the resulting machine will be regarded as obsolete when it is completed, unless, of course, the then current-generation processor can replace the original processor designed into the machine. Thus, in order for large numbers of microprocessor MIPS to be effectively harnessed into a large machine, it is crucial that the additional complexity associated with the machine interconnection hardware be kept as small as possible, or that the machine be built in a sufficiently modular way that it can accept upgraded processor chips.

Machines like RP3, which are designed to the characteristics of a particular microprocessor, and which require large investments in chip design, manufacturing, and packaging, have long development cycles and are ill-suited to compete against the increasing speed of commodity microprocessors. It is simply too hard to stay ahead of the RISC processor development curve. The memory interface of the microprocessor may not be sufficiently well architected to allow one to move along to next-generation chips in a shared-memory parallel processor. On RP3x, we are limited to a slow cycle time due to timing errors in the network interface chip, but even if we were not so limited, we could not switch to the faster CMOS ROMP without a redesign of the RP3x processor card.

Message-passing machines, on the other hand, have a much simpler and more modular interface between the processor and the interconnection network, and so are easier (and quicker) to build, and easier to upgrade when next-generation chips become available. For massively parallel applications and environments where the user is willing to completely rewrite programs to achieve a performance improvement of several orders of magnitude, these machines should be the vehicle of choice for parallel processing. For programs too large to be restructured easily, and for automatic parallelizing compilers, the future remains in shared-memory parallel processing with a modest number of processors.

### ***Concluding Remarks***

The RP3 project has been a large and ambitious project whose goal was to build a flexible research vehicle for studying hardware and software aspects of shared-memory parallel processing. RP3 is not a product, and was never intended to be a product; hence it should not be judged in comparison with product machines, but as a research vehicle. In that light we would argue that the project has been successful:

- The machine was completed and has been a reliable research tool for the past two years.



- We have been able to develop a version of Mach for RP3 that exploits the memory, cache, and processor architecture of the machine and lets programmers to use these facilities to create efficient parallel programs on RP3x. programmers use these facilities to create efficient parallel programs on RP3x.
- We have demonstrated that 40-way speedups are relatively easy to achieve on the 64-way prototype, and we have learned much about its use for parallel processing, as judged by the experience reported in this paper.

We hope that some of the lessons we have learned can be of help to others who are building similar machines, whether they are intended as research prototypes or product machines.

### Acknowledgements

Our work on RP3 could never have been completed without the assistance of many people, without whose conscientious efforts the system would not exist. In particular, we would like to thank Fran Allen, Gordon Braudaway, Wally Kleinfelder, Matt Thoennes, and Herb Liberman for their assistance and support during the RP3 project. Rajat Datta was responsible for the RP3 I/O service machines on VM; Rajat also contributed significantly to the first port of Mach to an RP3 PME. Tony Bolmarich was responsible for the implementation of EPEX for Mach/RT and Mach/RP3. The original port of Mach to the RP3 simulator was completed by Dan Julin of Carnegie Mellon University while he was a summer student working with the IBM Research Division. The Mach research group at Carnegie Mellon University, led by Professor Richard Rashid, has been very supportive of our work with Mach and their special efforts to make the latest versions of Mach available to us are greatly appreciated. Dan Renciewicz of IBM/TCS at CMU has been responsible for obtaining production releases of Mach for the RT system and making them available within IBM.

### References

- [1] M. Accetta et al., "Mach: A New Kernel Foundation for Unix Development," *Usenix Association Proceedings*, Summer 1986.
- [2] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante, "An Overview of the PTRAN Analysis System for Multiprocessing," *The Journal of Parallel and Distributed Computing*, vol. 5, no. 5, pp. 617-640, Oct 1988.
- [3] T. E. Anderson, B. Bershad, E. Lazowska, and H. Levy, Scheduler Activation: Effective Kernel Support for the User Level Management of Parallelism, Department of Computer Science, University of Washington, October 1990. Technical Report #900402.
- [4] J. Boykin and A. Langerman, "The Parallelization of Mach/4.3BSD: Design Philosophy and Performance Analysis," *Usenix Workshop on Experiences with Distributed and Multiprocessor Systems*, pp. 105-126, Fort Lauderdale, Florida, 1989.
- [5] W. C. Brantley, K. P McAuliffe, and J. Weiss, "RP3 Processor-Memory Element," *Proceedings of the 1985 International Conference on Parallel Processing*, pp. 782-789, August 1985.
- [6] R. M. Bryant, H.-Y. Chang, and B. S. Rosenberg, "Operating System Support for Parallel Programming on RP3," *IBM Journal of Research and Development*, Submitted for publication.
- [7] F. Darema, D. A. George, V. A. Norton, and G. F. Pfister, "A single-program-multiple-data computational model for EPEX/Fortran," *Parallel Computing*, no. 7, pp. 11-24, 1988.
- [8] J. Edler, J. Lipkus, and E. Schonberg, Process Management for Highly Parallel Unix Systems, New York University, 1988. Ultracomputer Note #136.
- [9] A. Gottlieb, B. D. Lubachevsky, and L. Rudolph, "Coordinating Large Numbers of Processors," *ACM TOPLAS*, January 1982.

- [10] D. N. Kimelman, "Environments for Visualization of Program Execution," *IBM Journal of Research and Development*, Submitted for publication.
- [11] T. J. LeBlanc, M. L. Scott, and C. M. Brown, "Large-Scale Parallel Programming: Experience with the BBN Butterfly Parallel Processor," *Proceedings of the ACM SIGPLAN PPEALS 1988—Parallel Programming: Experience with Applications, Languages, and Systems*, pp. 161-172, 1988.
- [12] K. Li and P. Hudak, "Memory Coherence in Shared Virtual Memory Systems," *Proceedings of the 5th Symposium on Principles of Distributed Computing*, pp. 229-239, August 1986.
- [13] J. Ousterhout, "Scheduling Techniques for Concurrent Systems," *Proc. of Distributed Computing Systems Conference*, pp. 22-30, 1982.
- [14] C. M. Pancake and D. Bergmark, "Do Parallel Languages Respond to the Needs of Scientific Programmers?," *IEEE computer*, pp. 13-23, December 1990.
- [15] G. Pfister et al., "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture," *Proceedings of the 1985 International Conference on Parallel Processing*, pp. 764-771, August 1985.
- [16] G. F. Pfister and V. A. Norton, "'Hot Spot' Contention and Combining in Multistage Interconnection Networks," *Proceedings of the 1985 International Conference on Parallel Processing*, pp. 790-795, August 1985.
- [17] M. Sejnowski, E. T. Upchurch, R. N. Kapur, D. P. S. Charlu, and G. J. Lipovski, "An overview of the Texas Reconfigurable Array Computer," *Proceedings of the 1980 National Computer Conference*, pp. 631-641, 1980.
- [18] R. H. Thomas, "Behavior of the Butterfly Parallel Processor in the Presence of Memory Hot Spots," *Proceedings of the 1986 International Conference on Parallel Processing*, pp. 46-50, August 1986.

# Experiences with Distributed Data Management in Real-time C<sup>3</sup> Systems

Paul J. Fortier

Naval Underwater Systems Center\*

David V. Pitts   John C. Sieg, Jr.   C. Thomas Wilkes  
University of Lowell†

## Abstract

Distributed real-time command, control, and communications (C<sup>3</sup>) systems are being used in many industrial and military applications. Real-time systems differ from conventional, general-purpose systems in that the consistency and correctness of the controlling process is dependent on the timeliness and predictability of the effects of the system on the controlled process. Real-time C<sup>3</sup> systems traditionally have not adequately addressed these requirements in a methodical fashion; systems have been hand-crafted and fine-tuned until they met testing requirements. In this paper, we review the evolution over the past three decades of the U.S. Navy's real-time C<sup>3</sup> system for its submarines. We note the lessons learned at each stage in the development of this system, and describe briefly a proposed architecture for the next generation of the system.

## 1 Introduction

Real-time C<sup>3</sup> systems control a physical system by the use of sensors, processing, and reactive devices. These elements collect information about the system under control, compute the state of the system, determine a reaction to the computed state, and invoke the proper stimulus to perform the desired control action. Real-time C<sup>3</sup> systems are being applied to a wide variety of physical systems, such as automotive

---

\* Author's address: NUSC, Code 2222, Newport, RI 02840. E-mail: fortier@nusc-ada.arpa

† Authors' address: Dept. of Computer Science, University of Lowell, One University Ave., Lowell, MA 01854. E-mail: {pitts, john, wilkes}@ulowell.edu



control, aircraft control, spacecraft control, power management, automated factories, and defense systems. The main function of these systems is to manage and utilize information. Therefore, these systems are nothing more than large information managers that must respond in predictable ways and within specified time frames.

These systems differ from conventional, general-purpose systems in that the consistency and correctness of operation of the controlling process is dependent on the timeliness and predictability of the effects of the system on the controlled process. For example, if a system is monitoring a reactor's core temperature and controls the flow of cooling fluid, it must not only process the incoming sensor data correctly and consistently, but also respond quickly to fluctuations and to boundary conditions in time to avert a catastrophic meltdown.

The stringent timing and interaction constraints of real-time  $C^3$  systems have a profound effect on the software and hardware architecture of these systems. A major notion concerning the design of such systems is predictability [Wat88]. Predictability is the ability to determine that the appropriate critical action will occur under all conditions. The system must be designed so that outcomes of operations and responses to events can be predicted ahead of time. This implies that the criticality and timeliness of real-time  $C^3$  operations must be defined. In practice, real-time  $C^3$  systems have not adequately addressed these requirements in a methodical fashion; systems have been hand-crafted and fine-tuned until they met testing requirements.

Since we regard real-time  $C^3$  systems as information managers, we focus on this aspect of their operation. In such a  $C^3$  system, information is repeatedly collected from the physical system. This information is sampled, converted, formatted, timestamped, and inserted into the control computer's database, once each sampling period of the sensors. This data must then be provided to the control software in order to be acted on to produce some desired control action.

Once the data is inserted into the database, it can be used to compute a variety of related parameters. For example, raw sensor inputs from a sonar system can be read from the device, and reduced to a bearing, range, and speed. These in turn can be used to compute detailed tracks, allowing for long-term tracking of an object. In addition, the raw information can be used to compute a potential profile for the object. This makes possible the classification and identification of the observed object.

To accomplish these and other tasks requires the database manager to store, manage, and retrieve data in real time, based on the critical needs of the system. The database manager cannot provide simple serial service to requestors as they arrive, since this is contrary to the prioritized, deadline-driven nature of processes (and therefore of transactions embedded in these processes). The database manager must provide services that allow for the dynamic selection of transaction steps for execution, and for the level of consistency and correctness based on the availability and timeliness requirements of real-time requests. Database management in a real-time environment must be tuned to the needs of the system based on the current phase of operations, not on the conventional, serializable execution sequences.

In this paper, we review the evolution over the past three decades of a real-time C<sup>3</sup> system developed by the U.S. Navy for its submarines. While describing the stages of the system's development, we focus on the lessons learned from experience with the various versions of the system. We conclude by describing briefly a proposed architecture for the next generation of this system.

## 2 Evolution of a C<sup>3</sup> Information-Management System

In this section, we give an overview of the evolution of the hardware architecture upon which naval submarine C<sup>3</sup> systems were built, drawn in part from the writings of Dr. A. J. VanWoerkum [Van87]. We also describe the parallel evolution of the software designs for the C<sup>3</sup> information-management system, in particular, the way the system uses, stores, and manages information in providing real-time support to the users of the system. This information is summarized in Table 1.

Table 1: The Naval Submarine Real-Time C<sup>3</sup> Systems—A Historical Perspective

System	Year in Production	Submarine Class	Hardware Architecture	Programming Language	DataBase Architecture
FCSMK113 Mod 2	1958	594,637	Fully analog	N/A	N/A
FCSMK113 Mod 6/8	1964	594,637,678	Hybrid analog-digital with digital control center	Assembler	N/A
FCSMK113 Mod 10	1973	686/687	Hybrid, only weapons analog	Assembler	"Resident regional" (shared in-memory database)
FCSMK113 Mod 10	1973	688 (Los Angeles)	Dual-bay UYK-7 Computer, analog data collection	ULTRA-32 (military macro assembler)	Resident regional
FCSMK117 Mod 0	1976	700/715	UYK-7, digital data collection, analog conversion and sensors	ULTRA-32 and CMS-2, a Fortran-like language that could call ULTRA-32 code	Resident regional
FCSMK117 Mod1/2	1976	594/637	Fully digital	ULTRA-32 and CMS-2	Resident regional
FCSMK118	1978	688	Fully digital	Fully CMS-2	Resident regional
BSY-1	1984-1986	688	Federated	Ada and CMS-2	Combination of resident regional and remote databases (file server)
BSY-2	Currently under development	SSN-21 (Seawolf)	Distributed	Fully Ada	Distributed autonomous Ingres

### 2.1 In the Beginning: Analog Computers

By 1964, the analog systems for controlling submersible naval platforms provided capabilities adequate for the threats of the day. However, they were large, complex systems which required expensive maintenance for minimal service. The addition of any new capabilities required physical floor space and balast, both of which were at

a premium and could not easily be given up for added capabilities. The systems required intensive man-machine interaction for operations and servicing.

In 1964, the U.S. Navy determined that the analog systems were inadequate to meet future needs, and could no longer provide the service necessary to meet changing missions and threats. To address this need, the Navy began to investigate digital computing systems as the avenue for future information processing and systems control needs.

## 2.2 Hybrid Analog-Digital Architecture

To alleviate the problems of analog computing technology, digital computing systems were considered as a means to provide improved real-time C<sup>3</sup> capabilities, reliability, maintainability, and operability, as well as to expand capability for on-board training and to reduce the size, weight, and manning requirements, and save money.

The first digital systems were required to use the then-standard Navy computer, its peripherals, a common programming language, and an existing real-time executive. In addition to replacing the functions of the analog computing system, further functionality was to be added.

The initial system was a hybrid that possessed numerous elements from the analog system, along with the new, central digital computer complex. The central computer complex replaced many of the tracking and navigation functions previously performed with the analog equipment. The central computer facility was linked to a variety of analog computing devices. These devices acted together to provide for tracking, navigation, environmental management, ship control, and weapon setting.

This class of real-time C<sup>3</sup> system represented the beginning of the era of information persistence. The previous analog-only system used information as it flowed in; there were no on-line storage management and retrieval capabilities. Information flowed into the analog computing devices, was acted on based on switch settings and circuit topology, and provided simple outputs. This was a hardware-only "massaging" of data to provide a result.

In the hybrid system, the central digital computer could sample, translate, store, and manage information as it flowed into the system. This provided the means to expand the capabilities of the overall system by supporting persistent storage of information, and the ability to compute a variety of derived information from the stored sensor-induced information.

### 2.2.1 Benefits and Limitations

The system was simple. It consisted of a single computer interfaced to the physical elements it was to control via analog-to-digital and digital-to-analog converters. The limitations also derived from the simplicity of this architecture. The central computer could only acquire and use data from the subset of sensors to which it was tied. The

full suite of sensors and information extraction devices were not available to the computer, and therefore it was limited to providing increased computational power for those sensors which were available.

The advent of data persistence, however, raised the consciousness of naval researchers. New C<sup>3</sup> algorithms arose that had not been conceived previously due to the dataflow nature of the early machines. The use of long-term time-tagged data led to further research in tracking, targetting, navigation, and environmental control algorithms. The simplicity of interaction and the limits on data input and output provided a unique laboratory within which concepts for interaction and control of physical real-time systems could be developed.

### **2.2.2 Lessons Learned**

The hybrid system indicated the usefulness of digital computers for the job of managing command, control, and communications within a real-time system. It led to a wide range of functional improvements and innovations that would not have been possible without the added computational power and the database storage, management, and retrieval capability.

## **2.3 Fully Digital Centralized/Multiprocessor Architecture**

A natural process of evolution led to the next architecture, which went into production around 1973. This architecture consisted of an all-digital central computing facility linked to the sensors and actuators by analog-to-digital and digital-to-analog converters. The remaining analog computational devices were replaced by digital computer hardware and software. This approach resulted in a computational platform that could be changed simply by recoding software, instead of by replacement of complex and expensive analog computers. In addition, the platform provided the first opportunity to digitally collect and store information from most of the ship's sensors and data collection devices. Only simple, stand-alone subsystems were not linked to the central computing facility during this transition.

The central computer complex was itself an early form of a multiprocessor/distributed system. Based on the Navy-standard UYK-7 computer, it consisted of dual processors interconnected by a shared memory bank and by interlocked input/output control computers. The input/output control computers were redundantly linked to all critical analog I/O devices to increase reliability. In addition, these two devices were cross-strapped to the two central computers to provide them with the ability to acquire and deliver data under a variety of fault conditions.



### 2.3.1 Database Architecture

The interesting feature of this system from a database standpoint was its use of a shared, in-memory central repository of encoded data, called the "resident regional." This repository was the site where all incoming data was stored initially, and to which requests for data were sent. Since the UYK-7 architecture was initially developed in the late 1950s, the memory was a small (196K) bank of non-volatile core memory. The data was encoded with (potentially) a number of information fields in a single 32-bit word. The fields could be as small as one bit, or as large as an entire word or multiple words.

### 2.3.2 Benefits and Limitations

The benefit of this approach was that for the first time it supported access of almost all important information in a digital form. This resulted in the research and development of a wide range of algorithms to aid in the ship's C<sup>3</sup> functions.

The problem with this approach was that it allowed use of the data by programs without mediation of a database manager. The ability of programs to set and use fields of varying sizes and encodings, along with the use of patches (on-line code fixes) and assembly code, allowed data to be altered without the knowledge of other programs. This resulted in a system that required extensive debug time for every change, and it was almost impossible to keep a database map accurate.

The limitations and problems of this approach far outweighed the benefits. The early systems were constantly crashing due to program errors that would erroneously alter individual fields or groups of fields due to miscalculated addresses or table bounds. The data corruption would cascade because programs using the tainted information would alter other parts of the database. The basic problem was lack of control over data access. The encoding scheme for information also proved cumbersome in that programmers were required to know about encodings when they constructed new functions. The initial system was developed with minimal controls, because of the fear that the controls themselves would cause failures due to increased overhead, which in turn would result in missed deadlines and unpredictable service sequences.

### 2.3.3 Lessons Learned

This hardware and database architecture provided the C<sup>3</sup> research community with a platform upon which to investigate and test concepts for real-time systems management in a realistic environment. However, the critical need for data access controls also became obvious. Experience with this system led to early research efforts on information management and real-time systems management.

## 2.4 Federated Architecture

From 1984 to 1986, the next-generation federated architecture was introduced [For79, San78]. The use of a federated architecture resulted from the need to increase processing and storage capacities while impacting neither the configuration of the central computer nor the ship's equipment footprint, and the need to deal with problems in response times, data senesence, and database management first encountered with the predecessor multiprocessor architecture.

The architecture for the federated system was developed utilizing the existing multiprocessor hardware as a base. This provided a means to retain the 2.6 million lines of source code developed for the previous system, yet allow for growth and change, which were both at a standstill in the fully-saturated multiprocessor system. The basic architecture consisted of the dual-processor central hub acting as the master unit, with up to eight peripheral slave processors embedded in the display consoles. These slave processors were VLSI implementations of the central computers, and were completely object-code compatible. The slave processors provided a means to recode some critical software, to remove load from the central master computers, and to distribute fragments of the database. In addition, the remote computer systems provided the opportunity to investigate alternative forms of database management.

### 2.4.1 Database Architecture

The database architecture in the federated system, illustrated in Figure 1, was based on centralized master copies of database objects, with shadow copies distributed to the peripheral sites. The peripheral sites operated on their shadow copies and sent updates to the centralized sites periodically to maintain the accuracy of the databases. The motivation was to make database objects more available to remote processes in order to increase concurrency of operations, thus increasing the ability of the system to meet the timing and dataflow needs of the real-time controller tasks.

The peripheral processors provided better-controlled database access by the use of a tabular structure, with controls over concurrent access to the data. However, they provided neither guarantees of real-time access, nor full concurrency control. They instead focused on yielding a simple, uniform data structuring and access scheme. This resulted in more readable and maintainable programs, and a means to control the complexity of the stored data and its uses. The interface between the centralized databases and the remote databases was handled by an interface process that translated from the old encoded fields to the expanded tabular structures.

### 2.4.2 Benefits and Limitations

This approach provided researchers with a means to correct some of the timing and data correctness problems of the earlier system, while allowing investigations into the benefits of distributed data and prioritized access to information.



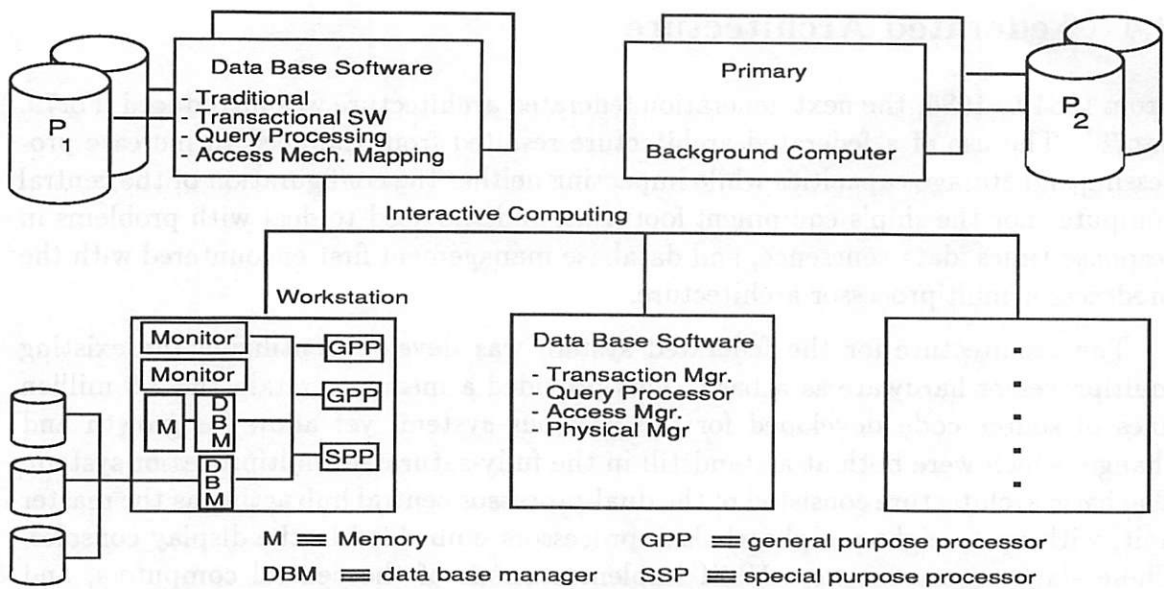


Figure 1: Federated System Database Architecture

However, the architecture fell far short of its promise in the development of policies and mechanisms that would be able to provide system-wide control, and still lacked keyed access to the data. How one could provide conventional database features such as concurrency control, integrity checking, transaction processing, and powerful query languages without suffering enormous time delays was still a mystery. Conventional database management features were seen as important for the future, but too expensive to include in an operational environment from a real-time systems perspective. Correct interaction and control of the database was still left up to the discretion of the programmers of the real-time C<sup>3</sup> software.

### 2.4.3 Lessons Learned

The major lessons learned at this stage were that database management had a place in real-time C<sup>3</sup> systems, and that future systems would require new techniques in order to use any of the features within conventional database management systems. In addition, distribution of data was an essential ingredient, as was the need to have database requests serviced based on the needs of the overall system, not on the needs of the database management system.

## 2.5 Distributed Architectures

There was a natural progression from the master-slave approach of the federated architecture to distributed architectures [Bur79, For86, For84]. The new designs allow decentralization of control, resulting in more highly available, reliable, and operable systems that can support incremental growth and change with little impact

on the system. The present thrust in naval computer developments is to more fully incorporate this technology into naval platforms. To realize these goals, research into fully-distributed operating systems and information management is in progress, along with incremental installation of distributed capabilities into present configurations, as well as in development projects such as the BSY-1 and the BSY-2 combat systems.

Current approaches begin to go beyond the "safe" paths of the past, and to introduce features of conventional data systems. For instance, the most recent system developments include the use of a unified query language as a means to enhance program development and maintainability, as well as to provide a unified database structure for the entire system. Distributed processing is seen as the means to provide processing power to the sites that need it most at any point in time. By the optimal placement of processing power and database tables, one can increase the performance of the overall real-time C<sup>3</sup> system. Delays associated with the transfer of information from site to site can be minimized, as can the delays associated with the invoking of the control action, which is the major task of the system. Distributed systems promise increased reliability, availability, modularity, fault tolerance, concurrency, and therefore performance and survivability, which are the key aspects of an operable real-time C<sup>3</sup> system. The system must provide predictable responses to external stimuli, fast enough to safely redirect its operations.

### **2.5.1 Architectural Approaches**

To date, there have been two attempts to build a distributed real-time C<sup>3</sup> system. The first attempt, called BSY-1, examined the use of a suite of global shared busses interconnected by bridges, as illustrated in Figure 2. The processors were distributed over the separate subbusses based on the types of processing they performed. For example, one subbus possessed the processors and processes to perform signal processing, another performed background functions, and yet another performed display processing functions. This separation of processing required that the flow of information be similar across each subbus, and limited the flow of information between subbus domains. Each of the domains had elements of the global data management function, while a separate subbus domain controlled the majority of database storage devices and thus the databases.

The second attempt, called BSY-2, is a distributed system developed around subdomains connected in a ring topology. This approach also strives to limit the volume of processors and processes on a domain, and to cluster processes in a domain based on the similarity of processing requirements and data needs of the processes.

### **2.5.2 Database Architectures**

In both BSY-1 and BSY-2, the approach has been to use conventional database management facilities and directly access information during critical time periods. In BSY-1, the database management system was essentially a global file system, with a

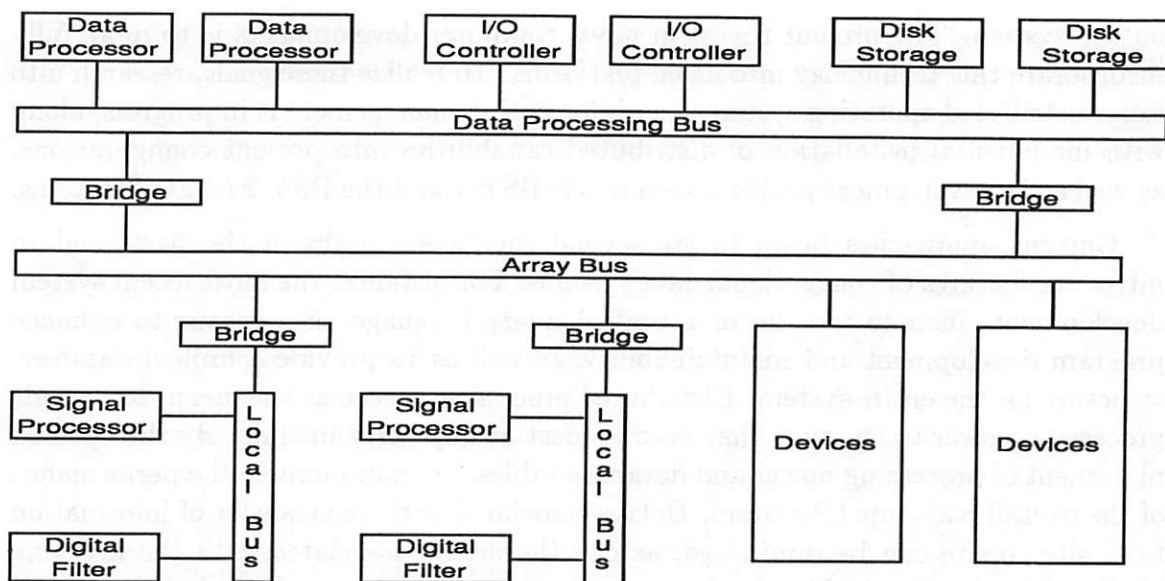


Figure 2: BSY-1 Bus Structures

user interface superimposed to give it the appearance of a full database system. This design allowed programmers to develop software around a common interface, and provided for the future insertion of a full-blown distributed database management system when one became available.

The database management system used in BSY-2 is more conventional. It uses off-the-shelf INGRES database systems distributed over the ring networks. The INGRES database systems do not attempt to synchronize updates or to cooperate on servicing queries. Users must know where data is stored and how to access the data across the network. The INGRES database systems act as isolated database managers, accepting queries and updates independently of each other. In addition to the INGRES databases, the system supports a distributed file server, which provides information directly to requestors. This supports real-time users who cannot afford the overhead of accessing the INGRES sites. The interface is a message-based facility, in which INGRES queries are all "pre-canned" as messages to a shell program which interprets the message as a one of a set of predefined queries.

### 2.5.3 Benefits and Limitations

In BSY-1, the separation of data storage facilities from data and array processing capabilities proved both a benefit and a drawback. Positive results included separation of functions and centralization of control. Added information management support for automatic data gathering and command decision aids provided further capabilities to the system. Increased redundancy added to survivability, fault tolerance and reliability. However, increased user overhead and limited growth resulted in bottlenecks.

The BSY-2 database system will provide the Navy with further experience with databases in a distributed environment. It will allow for the study of data flow in a fully distributed real-time system, and indicate the areas of the system that require fine tuning and more robust real-time access and management services.

#### **2.5.4 Lessons Learned**

The first distributed architectural approach (for BSY-1) was a database management system that fell far short of expectations due to problems associated with the file system and the interface. BSY-1's adhoc capabilities were found to degrade performance. This problem led to investigations of precanned and optimized queries which limited the systems flexibility, but maximized real-time performance.

BSY-2 and BSY-1 both still lacked an integrated system-wide database management system and architecture. Fragmented and isolated database management elements required database-query programmers to know about and access separate (and possibly different) DBMSs. At the time of this writing, precanned transactions appear to aid in the performance of the DBMS.

The BSY-2 is still under development. Studies indicate the need for a system-wide, integrated real-time database management system. Uniform access to data in the system would provide for ease of growth and change, for a more maintainable system, as well as for greater modularity and security.

### **3 Future Efforts**

In this section, we describe current proposals for next-generation hardware and software platforms for naval C<sup>3</sup> systems.

#### **3.1 Hardware-intensive, Massively Parallel Architectures**

Beyond the current distributed system architectures, efforts are under way to investigate and construct systems that allow the investigation of new information management techniques using massively parallel computations, and that rely on the extensive use of applications-specific integrated circuits (ASICs). We are developing a concurrency control mechanism supporting data management in a C<sup>3</sup> environment [For91]. The mechanism provides timely and predictable service to the most critical real-time transaction. The concurrency control mechanism will operate in a massively parallel, ASIC-based architecture, the Advanced Combat Systems Architecture (ASA).

Before describing the types and functions of the ASICs to be used in the system, we present the goals that guided the development of this architecture [For88]:

- Use commercially available components wherever possible, even at the expense



of functional performance.

- Provide a cost effective solution.
- Develop a reusable design philosophy through the use of standard cell libraries, combat-specific cell libraries, and commercial design systems and fabrication facilities.
- Support the evolution and possible transition of elements of the new architecture into existing systems.
- Achieve fault tolerance through replication rather than through reconfiguration, since reconfiguration can be as complex a task as the job the system is intended to perform.
- Use a modular approach to allow for individual functional alterations, and to facilitate certification and verification.
- Ensure extensibility to allow for unpredicted changes in the system requirements.
- Do not force the migration of a function to ASIC hardware if that function can be better performed on general-purpose hardware.
- Minimize the complexity of the architecture.

Our architectural approach is to treat ASICs as black boxes that accept input and produce a specified output. The functions provided by an ASIC are for the most part basic primitives required by the system. Collections of ASICs are built into the subsystems which comprise the system architecture. A subsystem consists of a collection of operationally related ASICs, along with a control element (action manager), an information management element, and a subsystem data object repository, all connected via an addressable switch interconnect. The subsystems are interconnected via multiple configurable switches.

The system supported by this architecture comprises, in part, a set of transactions on the system databases. Because of the nature of the C<sup>3</sup> environment with which we are dealing, the semantics, criticality, and periodicity of the transactions can be determined ahead of time. These transactions are analyzed off-line to provide a set of "transaction schemes" (graph representations of the transactions) that represent the transaction in the executing system (see the next section for more details). Because of the real-time aspects of the system, the criterion of database consistency enforced for schedules of system transactions is not serializability, but rather accounts for the criticality of some transactions. Also, requirements of the real-time C<sup>3</sup> database environment make compensatory measures, rather than the backward recovery found in most conventional database systems, more appropriate.

A key component of the analysis done on transactions is the identification of *atomic data sets* (ADSs) [SRL88]. An atomic data set is a set of database objects such



that the consistency of ADSs can be maintained independently. Thus, ADSs partition the database objects into disjoint sets with no consistency constraints between the ADSs. Knowledge of ADSs, along with the transaction semantics, is used to identify subtransactions that may proceed concurrently.

ASIC support for the system architecture occurs in three main areas. The first area is the traditional signal processing domain; the naval C<sup>3</sup> environment requires processing large amounts of information from sensors. Classification (What kind of object is this?) and identification (What specific object within the domain of this classification is this?) are some the functions supported by this set of ASICs.

The second area of ASIC support is the information management component of the architecture. Efficient processing of database queries is required, and ASIC support in the form of database accelerators is provided. Each database accelerator manages a portion of the entire distributed database in the C<sup>3</sup> system. The accelerator includes a controller processor that schedules primitive database operations (join, project, *etc.*) to specialized processors that perform those functions. The controller processor attempts to keep utilization of the specialized processors balanced.

The third area of ASIC support is transaction scheduling. Recall that each transaction has a specific criticality. The system must ensure that the most critical transactions are scheduled for execution regardless of serializability constraints and deadlines missed by less critical transactions. Work is proceeding on an ASIC that is designed to meet these scheduling requirements.

The required predictability of the functions and services that must be supported by the system has led to the migration of functionality from software into ASIC processors. In a system that had more *ad hoc* requirements on the services and functions needed at any instant, such a philosophy would be overly restrictive on the ability of the system to adapt. We foresee further migration of functions from software to hardware as our investigations continue. The naval real-time C<sup>3</sup> systems of the future will have architectures that heavily exploit heterogeneous massive-parallelism through the use of ASIC processors and components.

## 3.2 Integrated Database Management Architecture

The evolution of distributed real-time C<sup>3</sup> architectures described in this paper indicates the need for database techniques, such as transaction processing, within the real-time environment. Earlier architectures avoided the use of these techniques due to the time penalties associated with them. The goal of our current research is to develop an environment in which transaction techniques may be adapted to the needs of real-time processing. This effort is focused on developing scheduling and concurrency control algorithms that provide timely and predictable service to the most critical real-time transactions. The environment will be tuned to the needs of the most critical database transactions, but will also provide predictable service to other transactions. Transaction processing will have two major elements: a definition phase

and an execution phase.

In the definition phase each incoming transaction is examined and translated into an internal form, a transaction block graph. Block analysis, block materialization, and transaction optimization occur in this phase.

Using dataflow techniques, block analysis transforms each transaction into a transaction block graph. Each block defines a tightly bound sequence of database operations that are to be executed atomically.

Block materialization assigns blocks to logical sites. During the execution phase, a mapping of logical to physical sites may be generated dynamically based on site utilization and the location of data required by blocks.

Transaction optimization reorders the block execution sequence to provide a higher degree of concurrency. This is accomplished by transforming serial paths into parallel paths if no data dependencies exist.

In our system model, processes, transactions, task interactions, and timing (periodic, aperiodic, or triggered) are known ahead of time. Therefore, for most transactions, the corresponding transaction block graphs can be produced, optimized, and stored for later use, thus allowing the definition phase to be bypassed at run-time.

The execution phase applies a greedy scheduling algorithm and an optimistic concurrency control algorithm to the blocks produced by the definition phase. The execution priorities of blocks are dynamically reordered based on deadline weights, dynamic task criticality, and a static criticality based on the current state of system operation (*e.g.*, takeoff-cruising-landing of airplanes). Concurrency control is based on consistency predicates attached to atomic data sets; transactions not meeting their predicates are compensated for, and redone later if possible. Multiple levels of consistency, based on transaction semantics, are defined.

## 4 Summary

We have described the evolution of the naval real-time C<sup>3</sup> system used in submarines, from the use of specialized analog components to the predicted use of specialized digital components. For each of the major stages in this evolution, we have discussed the benefits and limitations of the approaches used at that stage, and presented the lessons that were learned. Finally, we outlined current work being done in the development of the next generation C<sup>3</sup> system. Our approach promotes a fully distributed approach to the issue of database management and the migration of functionality, at all levels, into hardware wherever feasible.

## References

[Bur79] W. K. Burke. FY82 CCS feasibility study. NUSC Working Document,

October 1979.

- [For79] P. J. Fortier. Simulation study of master/slave MCS MK-117. Technical Report NUSC TM# 79-2135, September 1979.
- [For84] P. J. Fortier. A reliable distributed processing environment for real-time process control. In *Proceedings of the Northeast Regional Conference*, Lowell, Massachusetts, 1984. ACM.
- [For86] P. J. Fortier. A real-time distributed operating system (RTDOS). In *Proceedings of the Third Workshop on Real-time Operating Systems*, Boston, Massachusetts, 1986. IEEE.
- [For88] P. J. Fortier. Advanced combat systems architecture. In *Proceedings of the First Joint Navy IRIED Symposium*, pages 431–445, John Hopkins University, Laurel, Maryland, June 1988.
- [For91] P. J. Fortier. *Transaction Processing in a Distributed Real-Time Control System*. PhD thesis, Computer Science Department, University of Lowell, Lowell, Massachusetts, 1991. In progress.
- [San78] T. Santos. Software development approach for utilizing a MK-81 display with embedded AN/UYK-7 evaluation. Technical Report NUSC TM# 78-2105, June 1978.
- [SRL88] L. Sha, R. Rajkumar, and J. P. Lehoczky. Concurrency control for real-time databases. Technical Report, Department of Computer Science, Carnegie-Mellon University, 1988.
- [Van87] A. J. VanWoerkum. Untitled writings on submarine combat systems evolution and future assessment. NUSC Code 101, 1987. NewLondon, Connecticut.
- [Wat88] P. H. Watson. An overview of architectural directions for real-time systems. In *Proceedings of the Fifth Workshop on Real-Time Software and Operating Systems*, 1988.





# The ION Data Engine

*Marc F. Pucci*

Belcore  
445 South Street  
Morristown, NJ 07960  
marc@bellcore.com

## *Abstract*

The ION Data Engine is a multiprocessor tasking system that provides enhanced data manipulation services for collections of workstations or other conventional computers. It is a back-end system, connecting to a workstation via the Small Computer Systems Interface (SCSI) disk interface. ION appears to the workstation as a large, high speed disk device, but with user extensible characteristics. By mapping an application's functionality into simple disk read and write accesses, ION achieves a high degree of application portability, while providing enhanced performance via dedicated processors closely positioned to I/O devices and a streamlined tasking system for device control.

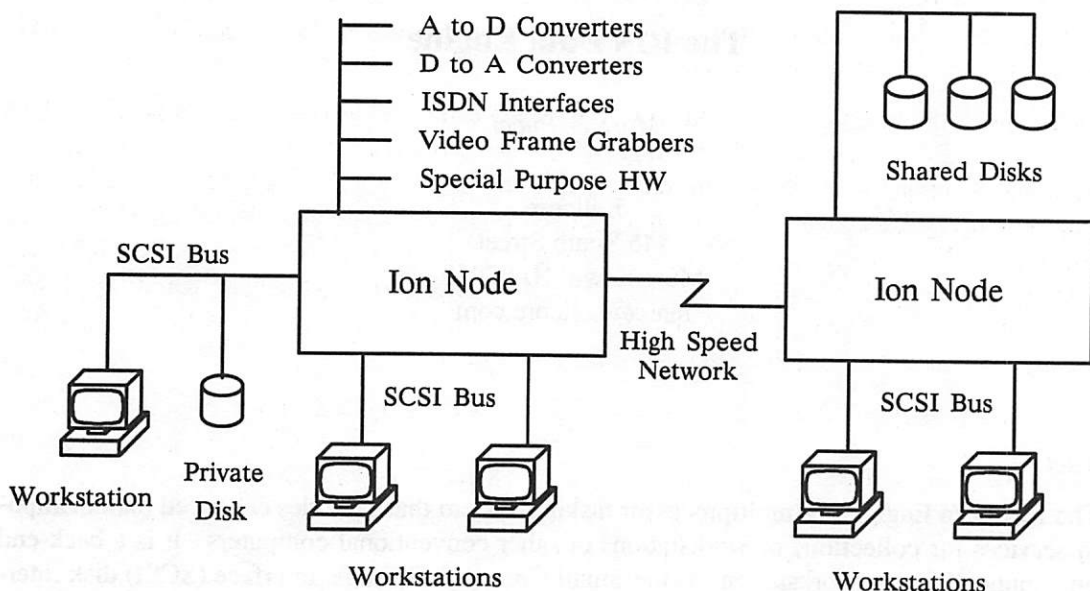
ION is being used as a platform for voice mail services in a user programmable telephone switch, and as a tool for measuring the I/O performance of computer-disk interfaces. Applications under development include an automated camera positioning system and an object repository.

## **1. Introduction**

The workstations that exploit the rapidly advancing state-of-the-art in processor technology can often be a bane to developers of applications that utilize dedicated special purpose hardware. An application that is tied to obsolete processor technology will soon suffer from comparative performance problems as newer workstation technology passes it by. However, interfacing new workstations to an existing hardware base is not simple. Initial workstation offerings often possess meager expansion characteristics, typically just a disk and network connection, so achieving even the electrical connection can be difficult.

Ideally, utilizing a new workstation should entail only simple recompilation of the application code; however, machine dependencies that result from the use of special purpose hardware complicate a code port. Workstation hardware may not be portable to different manufacturer's stations or even across a line of workstations from the same vendor. This can lead to the loss of a significant hardware investment as working components must be redesigned. Supporting multiple versions of hardware in order to preserve customer satisfaction with older configurations can also be expensive. Even hardware common to multiple stations, which is currently possible since many stations now offer VME bus interfaces, may still require device driver changes and must also track operating system variations from release to release.

An additional problem of using special purpose hardware on a conventional workstation is that the internal structure of the host operating system may not be conducive to the requirements



**Figure 1.** An ION system. Multiple ION nodes are connected via high speed networks. Each node connects several workstations and appears to be a large local disk.

of the hardware. It may be preferable to model an application into subtasks, each with its one flow of control; however, the relatively expensive context switch time for a general purpose operating system may make such an implementation infeasible for performance reasons. Also, the data rates generated by some hardware may have a detrimental effect on other functions in the workstation. In general, it is best to place compute power as close as possible to the source of data, passing only results or preprocessed information on to higher levels in the system. In this manner, devices requiring rapid response need not interfere with time-sharing operations.

ION addresses these problems by partitioning an application into hardware dependent and independent components, and providing a vendor independent interface between the two. The hardware independent components reside in the workstation, and are therefore easily ported to new architectures. The hardware dependent components are situated within a separate backplane-based environment, which is portable in its entirety across workstation changes. The low level connection between these components is the Small Computer Systems Interface (SCSI) [1] disk interface. Since each workstation accesses ION using its local disk system, which is a stable, well-defined interface, there is no need to change vendor supplied host system software. Current SCSI performance capabilities also provide a respectable (5 megabyte per second) data access rate.

ION configurations are expandable and sharable as needs dictate. Additional single board computers (SBC's) in a backplane can connect multiple workstations to the same set of hardware resources, or provide extra CPU cycles for I/O devices that require it. Further expansion is possible by using bus repeaters and local area networks to interconnect multiple ION nodes together. The basic structure of a large ION system is shown in Figure 1.

## 2. The ION Interface

A workstation sees ION as a local disk (an ION drive) with a data capacity of 2 terabytes. As such, it can contain random read/write data, traditional file system data, or more complex objects

for a variety of applications managed by sub-tasks running within the ION system. The latter is implemented by defining application specific functions, called *actions*, that are enabled by reading or writing specific disk block addresses within the ION drive.

For example, the interface to an analog to digital (A-to-D) conversion application within ION is implemented as an action defined on a set of 5 disk block addresses, each corresponding to 1 of the 5 analog channels available on the hardware. The controlling program that resides on the workstation merely reads an appropriate disk block to obtain the converted data (`lseek()` followed by `read()` in the Unix domain). By defining such interactions in terms of standard disk read and write accesses, the application remains portable across workstation changes, operating system releases, and to a large degree, complete operating system changes (e.g., Unix to VMS), while preserving any existing special purpose hardware investments.

A further advantage of the disk-like interface of ION is its robustness in the face of application failure. Since ION mimics a local disk drive, the worst case scenario for failure merely results in the apparent symptom that the ION drive has gone into an *off-line* condition – equivalent to a real drive losing power or spinning down. This should not have any long lasting effect on the workstation and is remedied by rebooting the ION system.

### 3. System Architecture

#### 3.1 Hardware

The hardware configuration of an ION node is shown in Figure 2. The current ION configuration uses the mature technology of high speed Motorola 68030 microprocessor based single board computers (SBC's). These CISC processors offer sufficient power for the current set of ION I/O devices and will be upgraded to faster RISC or CISC processors when more demanding peripherals are in use. An SBC is dedicated to each workstation connection, primarily because most hosts insist on using the same SCSI bus address. Additional disk interfaces are used to control local node storage, which may consist of file system data or application specific object repositories. Large buffer memory, on the order of hundreds of megabytes, is used as a cache for physical device data. The high speed network interface connects an ION node to other similar nodes in a system. ION currently uses Ethernet for this purpose, and will use a faster Bellcore developed metropolitan area network interface in the future.

#### 3.2 Software

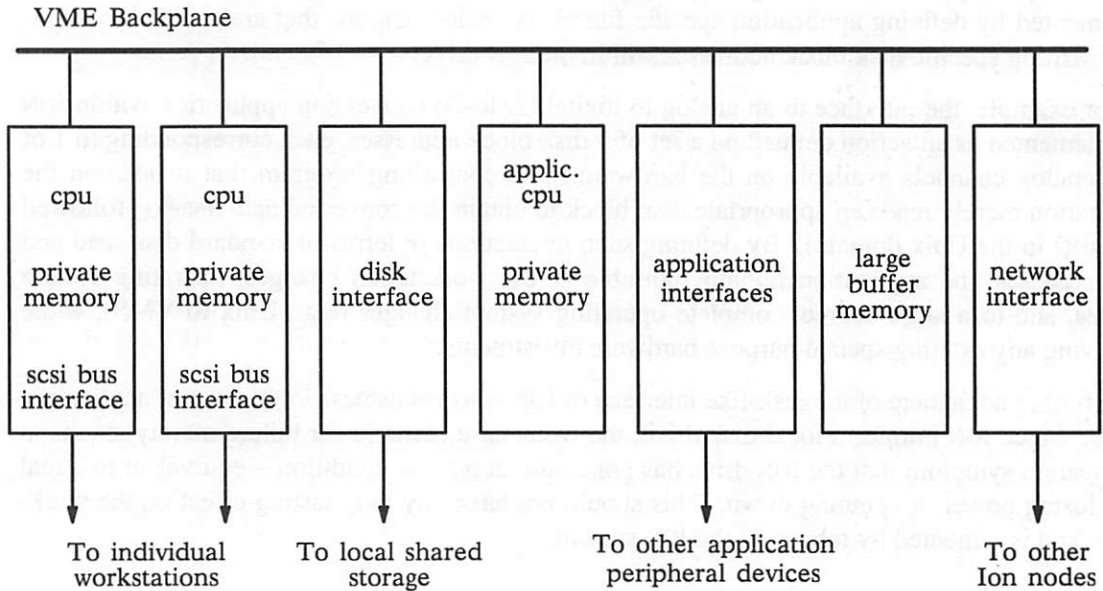
ION is implemented as a fast tasking system, similar in scope to such minimal kernels as Alpha [2], Arts [3], Chaos [4], Mach [5], Ra [6], Spring [7], Synthesis [8], V [9] and others. However, ION is specifically geared towards supporting peripheral devices: it adds flexible processing power to their functioning, and then interfaces the resultant modified, intelligent peripheral to other conventional computers in an easy and portable manner. All system and application software are memory resident with their own flow of control and execute in the same address space. Modularity is achieved by dedicating tasks to specific system functions, and passing requests for service through client-server transactions on the same or other ION processors.

---

Unix is a registered trademark of AT&T Bell Laboratories.

VMS is a registered trademark of Digital Equipment Corporation.

Ethernet is a registered trademark of Xerox Corporation.



**Figure 2.** An ION node. Each node contains a dedicated single board computer (SBC) to manage each workstation interface. Other SBC's control local storage, manage object repositories, control additional I/O devices, or run application code.

While state machines rather than multiple tasks are often used for managing disk-like operations, the recursive state machines necessary for SCSI are difficult to design and enhance. Alternatively, assigning a task to the management of each individual workstation connection and I/O device simplifies the coordination of multiple objects, which in turn allows for easier parallelization of I/O activities. Individual SCSI tasks manage their own disconnect/reconnect behavior on the SCSI bus on a device by device basis. The multiple flows of control offered by the tasking system are useful for application as well as system functions. Such operations as consistency management, network control and routing, and recovery management are more easily designed as separate tasks. Multiple processors fit naturally into such an environment and provide needed power and responsiveness for handling multiple powerful workstations and devices.

While acknowledging that multiple tasks can lead to a loss of performance [10], ION alleviates this by exploiting certain characteristics of its environment: With a single address space and no need for the complete functionality of a general purpose operating system, many optimizations are possible. Speed is the paramount requirement for the ION tasking system, and therefore task switching, event synchronization and interrupt response time have been designed with minimal overhead. Table 1 summarizes some of these characteristics. The cost of using a tasking system over a conventional state machine can be seen in some of the performance measurements presented in Section 5.



Interrupt dispatch time	4 $\mu$ s
Null interrupt service time	9 $\mu$ s
Task switch	25 $\mu$ s
Event synchronization	16 $\mu$ s
Simple system call	8 $\mu$ s
Same processor null client/server interaction	105 $\mu$ s
Remote processor null client/server interaction	140 $\mu$ s

**Table 1.** ION system characteristic measurements.

Interrupt dispatch time is the delay between when an I/O device signals its need for service and when its interrupt service routine is entered. Null interrupt service time is the time needed to save and restore pre-interrupt state and increment a counter. The task switch time measures the delay incurred when one task suspends and a second task continues execution without any specific form of synchronization. It is mostly the time required to save and restore 2 sets of the general purpose registers of the 68030. Event synchronization time must be added to the basic task switch time when 2 tasks synchronize through the message queueing and dequeuing primitives. A simple system call is similar in timing to the null interrupt time, since much of the same functionality must occur.

The null client/server interactions are essentially remote procedure call [11] (RPC) interfaces between cooperating tasks. When on the same processor, this involves task-switching the receiving and sending tasks, queueing and dequeuing the request and the response message, and determining the location of the sending and receiving queues. When the RPC crosses processor boundaries, the timing includes the single processor case above plus interrupt latency for the sending and receiving message and extra interrupt processing necessary in the processor-to-processor communications functions. (A single interrupt indicates message reception from multiple processors in the system, so a number of input sources must be checked for the presence of a message.)

### 3.3 Internal ION System Services

ION is a minimal kernel consisting of a set of simple primitives for constructing applications. Applications are composed of one or more tasks, with each task running till completion or resource blockage. The system primitives include:

<i>new_task</i>	Create a new flow of control. Execution begins at a supplied function address. Tasks can be created at interrupt time.
<i>task_exit</i>	Destroy a task.
<i>target_handler</i>	Define the set of disk block addresses to which an ION application will respond. Also define the action function to invoke when a workstation accesses a block from the set.
<i>scsi_dma_read</i>	Obtain data from the workstation across the SCSI bus. Used within an ION action routine to satisfy a workstation <i>write disk block</i> command.
<i>scsi_dma_write</i>	Return data to the workstation across the SCSI bus. Used within an ION action routine to satisfy a workstation <i>read disk block</i> command.
<i>scsi_disconnect</i>	Disconnect from the SCSI bus without completing the current workstation <i>read</i> or <i>write disk block</i> command. Used if the action cannot complete immediately and the workstation's operating system supports SCSI disconnection.

<i>scsi_reconnect</i>	Counterpart to <i>scsi_disconnect</i> . Reacquire the SCSI bus across long operations in order to continue or terminate a workstation command. Use of disconnect/reconnect will improve SCSI bus utilization when other devices share the same bus.
<i>queue, dequeue</i>	Queue and dequeue messages. This is the only task synchronization facility available in ION, essentially combining task activation with data availability, rather than supplying separate primitives for each. Dequeueing a message from an empty queue causes task suspension (a non-blocking variant also exists).
<i>attach_intr</i>	Define the application routine to be invoked when a hardware device generates a request-for-service interrupt.
<i>memory_alloc</i>	Allocate system memory. Three types are available: cached, uncached, and VME memory. Cached memory is traditional system memory that can be cached by the processor's memory system hardware for faster access. Uncached memory is used for regions of local memory that can be changed by I/O devices or external bus references. This class of memory is necessary for SBC's that do not have snooping caches. VME memory is the pool of external memory available for buffering large quantities of device data.
<i>memory_free</i>	Return the above allocated memory to its corresponding pool.
<i>copy_block</i>	Move a block of memory from here to there. On systems with hardware data movers or internal DMA controllers this may not be a CPU copy loop. Used when an I/O device cannot DMA out of VME memory, but must copy data to its private memory first.
<i>time_out</i>	Define a function to be called with supplied arguments in a certain amount of time. Time granularity is 1 millisecond.
<i>get_utimer</i>	Return a backplane-synchronized microsecond timer for performance measurements or time stamping of data.
<i>perf_snap</i>	Generate a performance profile of application activity showing subroutine execution time percentages. While not truly a primitive for application construction, it has proved to be very useful for application development.

### 3.4 Why SCSI?

The Small Computer System Interface, SCSI, is a high-level device interface standard. It exploits the use of programmed intelligence within each device on the bus, offloading many functions otherwise performed by the host. SCSI is the fastest common interface to a variety of workstations. None of the design issues in ION are constrained to SCSI, and its use in future versions of the system will be reevaluated when new interfaces, such as FDDI, mature and are commonly available. In the interim, SCSI provides a fast, flexible expansion interface with a defined next generation architecture (SCSI-2) offering significantly higher performance. Additional details on SCSI can be found in the appendix.

#### 4. An Example Application – Analog to Digital Conversion

ION provides the platform for analog to digital (A-to-D) services for a voice messaging application of a programmable telephone switch system called GARDEN. It provides the physical interface to readily available VME cards, and also provides additional processing power to off-load the interrupt handling and data formatting necessary for their operation. It provides a degree of protection against obsolescence, since the hardware investment in these peripherals will be portable to the expected next workstation upgrade. Additionally, since the hardware dependent A-to-D code remains within ION, no driver changes are necessary upon workstation upgrade.

Similar commercially available solutions to such problems are now appearing on the market [12]. ION offers the advantage of user programmability of the interfaces and device characteristics. This leads to greater functionality and power located off the host processor and in the peripheral device.

The part of the A-to-D application that resides within ION is structured around three cooperating tasks. One task is activated by periodic interrupts from the hardware and extracts the raw data from the converter, placing it into a queue for temporary storage. Since the data extraction is not done at interrupt time, less system activity occurs at a high CPU priority level. The interrupt routine and the task share a pair of queues and a token which is passed between the queues to coordinate activity. This prevents the interrupt routine from reactivating the task if the task has not completed its previous data extraction.

The second task is a generic system utility that translates 16-bit linear data into 8-bit mu-law data, as required by this particular application. It is essentially performing data compression on the input stream.

The third task interfaces to the SCSI bus and returns data to the workstation when requested. This task defines a SCSI action function which contains 4 block addresses for each of 5 A-to-D channels. Each channel contains a block address to start conversion, stop conversion, return status, and retrieve A-to-D data.

The part of the application that runs on the workstation requests converted data in response to a start/stop signal from other system hardware, which indicates the beginning and end of a recording session. Upon start, the workstation reads the A-to-D start address for an appropriate channel, activating the device. It then retrieves data by reading the data block address for that channel, while also monitoring for an end-of-session indication. When the latter occurs, the workstation reads the stop address, halting the data conversion. It continues to read the data address until all buffered data have been obtained. The channel is then available for reuse.

##### 4.1 SCSI Flow Control

An underrun condition occurs if the workstation requests data from a channel without any data. At this point, two alternatives exist: The application can suspend the host's I/O operation until data are available, or it can return immediately with some indication that the workstation program should reattempt the data request at a later time. The latter alternative is essentially polling, which can be inefficient and decrease SCSI bus utilization. However, waiting for the data to be available will tie up the workstation's channel into ION, making it impossible for other applications to communicate over the SCSI bus. (Only the ION connection is affected, other SCSI devices are still accessible.)

The above problem can be mitigated somewhat by using 1 of the 8 logical unit numbers (LUN's) defined by SCSI for sub-device access, effectively giving 8 independent channels into

ION. However, not all workstations support multiple LUN's. The problem of limited channels into ION is solved with SCSI-2, which includes a *tagged command* facility that allows multiple outstanding commands to be issued to a single target device. Both the workstation and the target are responsible for remembering that multiple jobs are pending, and properly coordinating the returned information across the SCSI bus.

## 5. Using ION to Measure Processor-to-Disk I/O Performance

Another application of ION is a measurement tool for studying the SCSI interface performance of a computer system. Contrary to measurement systems such as IOSTone [13] and IOBench [14], which use synthetic workloads as a basis, or trace driven studies that require system modifications [15], these measurements are made from the perspective of the disk device, not the workstation, and therefore reflect hardware capabilities, not software characteristics. Also, the procedure used does not require any changes to the host systems. The data thus captured can be used to optimize software performance within the operating system or guide the design of data access routines in a sophisticated user application.

The performance measurement system is defined as an action over a large range of block addresses, corresponding to the "A" section of a standard raw disk device partition table. The action function records the time and type of SCSI state transitions, and the amount of data transferred, and returns or accepts host data immediately. No intermediate SCSI disconnection occurs. From this information, throughput, transfer rate and overhead calculations can be made. Measurements were taken on 3 sample workstations, referred to as Systems A, B and C\*. Each workstation is connected to an ION system in an idle environment. For contrast, comparison to a second ION system is also shown. Unless indicated otherwise, all measurements are taken using instrumented code running only in the ION system and initiated through the raw disk system interfaces provided by each workstation.

The timers used for each measurement are triggered by the interrupts that correspond to phase changes of the SCSI protocol (each SCSI command can be composed of multiple instances of 6 types of information exchanges called phases). Hence, such measurements are not influenced by operating system overhead, which is subject to considerable variation between vendors. The systems were operated in the asynchronous SCSI data transfer mode, as this was the only mode of operation common to all 3; only one workstation supported synchronous transfers at the time of the experiments, which would improve the relative performance of that station. However, data transfer is only one part of the more complex SCSI command protocol. The discussion measures performance in megabytes per second (mbs), kilobytes (kb) and milliseconds (ms).

### 5.1 I/O Transfer Rates and Command Time

Figure 3 illustrates the read data transfer rate of the 3 workstations and ION. The abscissa is indexed in disk sectors of 512 bytes, which corresponds to the physical block size used on most SCSI drives. Most Unix file system traffic occurs in 16 sector increments. This measurement shows only data transfer rate, and does not include any additional SCSI command overhead. It is

---

\* The workstations employed in this exercise were selected for the sole purpose of illustrating this application of ION. The exercise was not intended as an exhaustive or scientifically precise analysis of computer products. The results reported herein are merely examples of results achieved and should not be considered as either positive or negative judgements about any product or vendor.



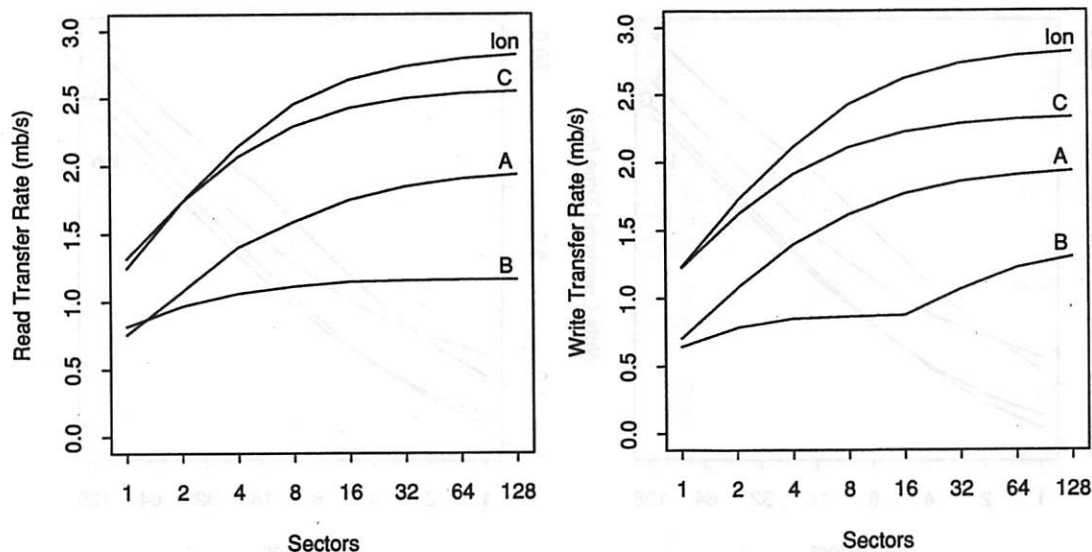


Figure 3. I/O transfer rates. These plots do not include other SCSI phase overhead.

therefore indicative of the maximum performance attainable by each system. As transfer size increases, the performance of System B flattens at about 1.2 mbs. System A reaches its maximum rate at 1.9 mbs, while System C transfers attain a maximum rate of 2.5 mbs. The ION drive is seen to reach 2.8 mbs.

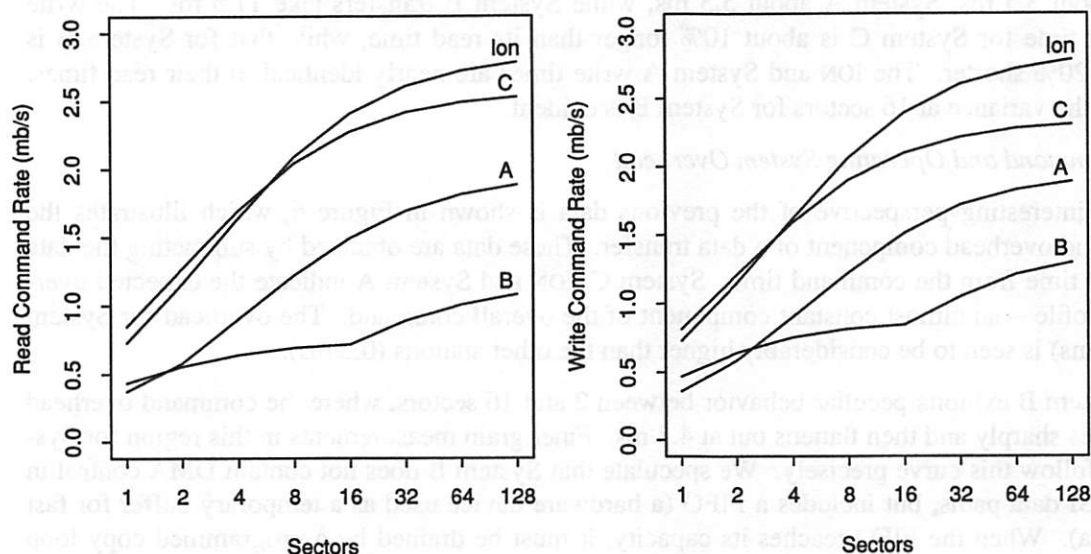


Figure 4. Command transfer rates. These plots include other SCSI phase overhead.

Overall command transfer rates are lower, as shown in Figure 4. These are seen to reach asymptotically the rates of the previous figure, as overhead becomes less of a percentage of the entire command time. Note the abrupt change in the behavior of System B at 16 sectors. This is more apparent in a later figure showing only the overhead component of a command.

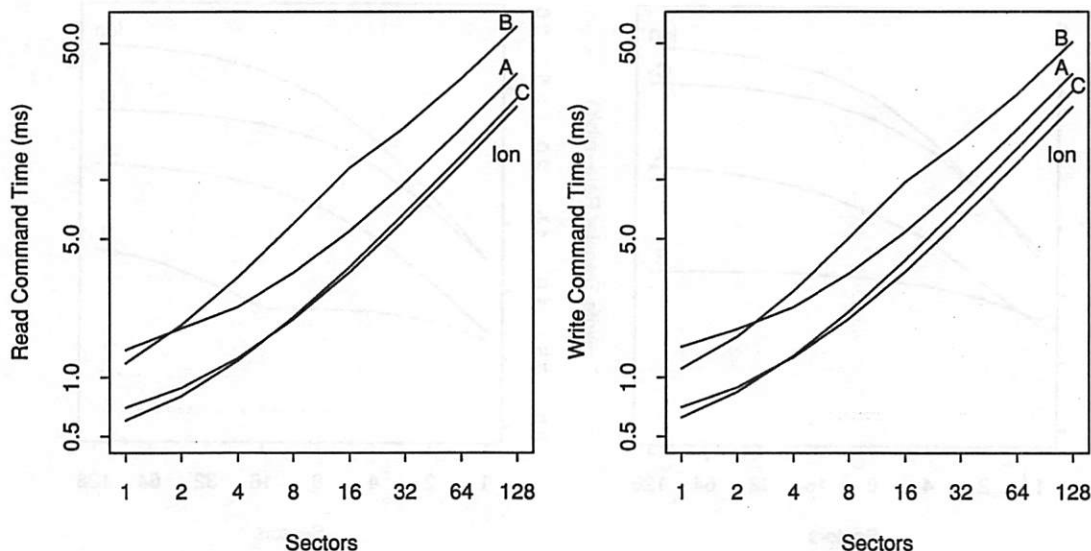


Figure 5. Data transfer command time.

Figure 5 measures the time for the entire I/O command, including the data transfer component. As expected, overall time for each command increases with the transfer size but with different break points for each system. Single sector transfers take between 0.5 and 1.5 ms for all systems (a time period generally dwarfed by physical disk access characteristics). However, performance at 16 sectors covers a much wider spectrum. Read transfer times on ION and System C take about 3.5 ms, System A about 5.5 ms, while System B transfers take 11.5 ms. The write transfer time for System C is about 10% longer than its read time, while that for System B is almost 20% shorter. The ION and System A write times are nearly identical to their read times. Again, the variance at 16 sectors for System B is evident.

### 5.2 Command and Operating System Overhead

An interesting perspective of the previous data is shown in Figure 6, which illustrates the command overhead component of a data transfer. These data are obtained by subtracting the data transfer time from the command time. System C, ION and System A indicate the expected overhead profile – an almost constant component of the overall command. The overhead for System A (0.8 ms) is seen to be considerably higher than the other stations (0.2 ms).

System B exhibits peculiar behavior between 2 and 16 sectors, where the command overhead increases sharply and then flattens out at 4.3 ms. Finer grain measurements in this region for System B follow this curve precisely. We speculate that System B does not contain DMA control in the SCSI data paths, but includes a FIFO (a hardware device used as a temporary buffer for fast I/O data). When the FIFO reaches its capacity, it must be drained by a programmed copy loop executed by the processor, thereby increasing the overhead associated with larger transfers.

### 5.3 State Machines v. Multiple Tasks

Examination of all the plots for System C and ION usually shows ION as a slightly faster device; however, in Figure 6 System C is consistently faster. The explanation involves the different methods in which the SCSI protocol is implemented in each system. Most of the additional overhead for a transfer is caused by the phase changes (and their accompanying interrupts) in the SCSI protocol. These changes can be handled more rapidly with a state machine implementation

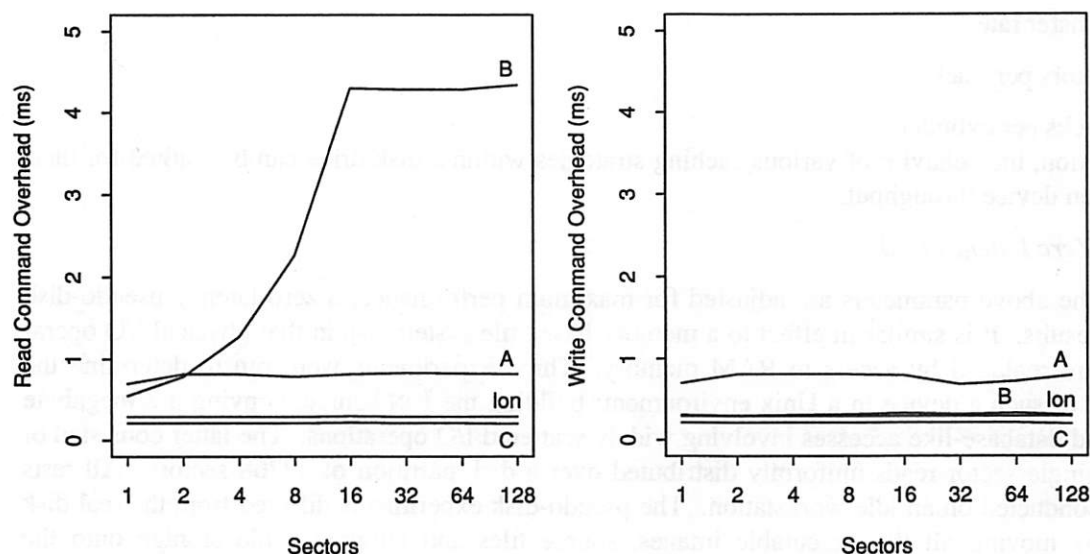


Figure 6. Command overhead. These plots illustrate command time less data transfer time.

(typical of most Unix systems) than by the multiple tasks in use in the ION system. In a state machine, the phase transitions can be controlled at interrupt time, while ION must incur the overhead of task synchronization and task scheduling before the transition can occur.

System A	715 $\mu$ s
System B	245 $\mu$ s
System C	261 $\mu$ s
ION	265 $\mu$ s

Table 2. Test-unit-ready time. This is a simple SCSI command without any data transfer.

All 3 workstations generate additional SCSI command sequences during the course of normal system operation. One such command is the *test unit ready* command, which is used to verify that a physical device is operational. It is a minimal command, and contains no data transfer component. Most systems issue this command at device *open* time. Table 2 illustrates the time required by each station for this simple command. System A requires twice the time as the other stations and may indicate either excessive interrupt response time or a slow SCSI controller chip. It is consistent with the earlier command overhead plot.

## 6. A Memory-Based Pseudo-Disk Application

Another ION application is to function as a pseudo-disk for analyzing computer system behavior as disk technology changes. By defining hardware disk characteristics in software, it is possible to use ION to study the impact of new disk technology before it is commercially available. When such an application is backed by sufficient buffer memory in ION, actual file-system behavior, rather than simulation, can be monitored.

Using software running within ION, the following attributes can be controlled:

- Rotational latency

- Head positioning time
- Transfer rate
- Sectors per track
- Tracks per cylinder

In addition, the behavior of various caching strategies within a disk drive can be studied for their affect on device throughput.

### 6.1 A Zero Latency Disk

If the above parameters are adjusted for maximum performance, a zero-latency pseudo-disk drive results. It is similar in effect to a memory based file system [19], in that physical I/O operations are replaced by access to RAM memory. Three experiments were run to determine the impact of such a device in a Unix environment: building the ION source, copying a 2 megabyte file, and database-like accesses involving widely scattered I/O operations. The latter consisted of 4000 single sector reads uniformly distributed over a disk partition of 32768 sectors. All tests were conducted on an idle workstation. The pseudo-disk experiments differed from the real disk case by moving all the executable images, source files and temporary file storage onto the pseudo-disk.

Operation	Real	User	System	Improvement
Build ION – Real Disk	156.7	81.2	28.2	
Build ION – Pseudo-Disk	126.6	81.0	27.7	1.2
Copy File – Real Disk	11.5	0.0	1.6	
Copy File – Pseudo-Disk	7.2	0.0	1.6	1.6
Random Access – Real Disk	67.7	0.1	4.2	
Random Access – Pseudo-Disk	8.1	0.1	2.7	8.4

Table 3. Real disk v. pseudo-disk performance. Time measured in seconds.

The results indicate that this type of pseudo-disk is not very practical for single-user operations involving sequential file access. The read-ahead/write-behind [20] strategies employed in Unix work exceptionally well when the CPU performs some processing of the data between I/O accesses. A better improvement is seen when only minimal processing occurs, as in the file copy. For random behavior, the pseudo-disk is seen to behave significantly better than a conventional disk drive. Further study is necessary to compare such results when multiple users are involved, for example, on a file system server. When requests for sequential file access occur from multiple sources, it is possible that the resultant behavior more closely resembles scattered accesses as requests are intermixed.

## 7. Conclusions and Future Work

ION is far from being a completed project. The system is evolving continuously as it accommodates additional peripheral devices and application functions. Ethernet interfaces are being added to improve the physical accessibility of ION nodes by other hosts. ION has proved to be a flexible tool for experimenting with new hardware, especially given its nonintrusive (to the host workstation) development environment.

The programming model for ION has evolved since its initial use in the voice messaging system. The original primitives used for application construction were too restrictive and associated excessive knowledge of the SCSI interface in application code. The current design uses a data



queuing model with a programming interface more conducive to "action at a distance" control of peripheral behavior; it more closely resembles a large-grain data flow system [21] [22]. As an example of the new design, about a dozen lines of user commands are sufficient to define an application that manipulates multiple streams of CD quality digitized audio data from a host. This configuration defines a scsi disk block address for each stream, level converts, mixes, compresses and rate adjusts the data, and transmits the output over a broadcast Ethernet UDP port.

## REFERENCES

1. ANSI X3.131, Small computer systems interface (SCSI), American National Standards Institute, Inc.
2. J. Northcutt, Mechanisms for reliable distributed real-time operating systems - the Alpha kernel, Academic Press, 1987
3. H. Tokuda and C. Mercer, ARTS: A distributed real-time kernel, *Operating Systems Review*, 23(3):29-53, July 1989.
4. P. Gopinath and K. Schwan, Chaos: Why one cannot have only an operating system for real-time applications, *Operating Systems Review*, 23(3):106-125, July 1989.
5. M. Accetta et. al., Mach: a new kernel foundation for Unix development, *Proc. Usenix Summer Conference*, Atlanta, GA, 1986.
6. C. Wilkenloh et al., The clouds experience: building an object-based distributed operating system, *Proc. Distributed and Multiprocessor Systems Workshop*, 333-347, Fort Lauderdale FL, October 1989.
7. J. Stankovic and K. Ramamritham, The design of the Spring kernel, *Proc. Real Time Systems Symp.* 146-155, CA, December 1987.
8. C. Pu, H. Masselin and J. Ioannidis, The Synthesis Kernel, *Computing Systems*, 1(1):11-32, Winter 1988.
9. D. Cheriton, The V kernel: a software base for distributed systems, *IEEE Software* April 1984.
10. D. D. Clark, The structuring of systems using upcalls, *Proc. 10th Symp. on Operating Systems Principles*, Washington, December 1985.
11. A. Birrel and B. Nelson, Implementing remote procedure calls, *ACM Trans. on Computer Systems*, 2(1):39-59, Feb. 1984.
12. Desklab<sup>TM</sup> Real-time Analog Data I/O, Gradient Technology, Inc. Burlington, NJ.
13. A. Park, J. Becker and R. Lipton, IOStone: A synthetic file system benchmark, *Computer Architecture News*, 18(2):45-52, June 1990.
14. B. Wolman and T. Olson, IOBench: A system independent IO benchmark, *Computer Architecture News*, 17(5):55-70, 1989.
15. J. Ousterhout et. al., A trace driven analysis of the Unix 4.2 BSD file system, *Proc. 10th Symp on Operating System Principles*, December 1985, Washington.

19. M. McKusick, M. Karels and K. Bostic, A pageable memory based system, *Proc UKUUG*, London, Summer 1990.
20. D. Ritchie and K. Thompson, The UNIX timesharing system, in *Bell System Technical Journal*, vol. 57, no. 6, part 2, July-August 1978.
21. R. Babb II, Parallel processing with large-grain data flow techniques, *IEEE Computer*, 17(7), July 1984.
22. R. Rasmussen et. al., Max: Advanced general purpose real time multicomputer for space applications, *Proc. Real Time Systems Symposium*, San Jose, CA, December 1987.

## 8. Appendix – What is SCSI?

SCSI, the Small Computer System Interface, is a protocol definition for connecting processors, disk drives, printers and other devices. It is a high-level interface that expects a significant amount of intelligence within the controller associated with each device. This is in sharp contrast to other disk interfaces (e.g., SMD) where individual devices typically respond only to control signals, and all programmed intelligence resides in the host controller. Up to 8 devices can exist on a single SCSI bus, each taking a fixed SCSI device identifier number. Most hosts insist on being device 7. Each device can be composed of up to 8 independent subdevices using a logical unit number (LUN) facility. However, few devices and operating system implementations support this feature. The maximum bus length is about 20 feet, although a differential bus specification also exists which permits a total bus length of 80 feet.

### 8.1 SCSI Devices, Commands and Phases

SCSI devices are typically classified as either initiators or targets, although these roles need not be permanent. As the names imply, an initiator (usually the host processor) starts an operation by arbitrating for the SCSI bus and selecting a target device (such as a disk drive) to respond to its request. All further action is controlled by the target device which indicates its intentions by changing the SCSI bus phase.

A facility known as *disconnect/reconnect* allows better utilization of the SCSI bus. If a command involves a relatively long delay before requested data will be available, the target can disconnect from the SCSI bus, making it available for other targets, and reconnect when the data are ready. Such delays are normally encountered during physical head repositioning on disk drives. The initiator informs the target of its ability to accommodate this behavior during a message exchange before the actual command begins.

Six phases are defined by the SCSI specification to coordinate transmissions between the initiator and the target. The terminology of *in* and *out* used below is always with respect to the initiator. All phase changes are controlled by the target.

<i>command</i>	The target is requesting a multibyte command sequence that defines the desired operation.
<i>status</i>	The target is returning a single status byte to the initiator indicating the outcome of the command.
<i>message in</i>	The target is sending a control message to the initiator. Messages are transmitted to indicate parity error detection, command completion and identification of sub-units within a target.

<i>message out</i>	The target is requesting a message from the initiator. This phase is usually generated in response to a control signal ( <i>attention</i> ) asserted by the initiator.
<i>data in</i>	The target is instructing the initiator to begin accepting data as a result of the command.
<i>data out</i>	The target is requesting data from the initiator, as described by the command.

## 8.2 Whither SCSI?

The next generation of SCSI, SCSI-2, is a mostly upwards compatible change, with many optional SCSI-1 commands and messages becoming mandatory. The significant improvements involve the width of the data path and the cycle time for each individual transfer on the bus. SCSI-1 has an 8 bit data path with a minimum cycle time of 200 nanoseconds yielding a maximum throughput of 5 mbs. SCSI-2 can use optional secondary cables, providing 16 or 32 bit transfers. In addition, the minimum transfer cycle time is reduced to 100 nanoseconds. Hence, the maximum throughput is 40 mbs. SCSI-2 peripherals and controllers are beginning to appear on the marketplace.

SCSI-2 also provides a mechanism for command queueing, where an initiator can send multiple commands to a target, allowing it to service these requests in a device specific optimal ordering. A further feature allows a target to inform an initiator of a change of condition, even if the initiator does not have a command pending with the device. This is instrumental in returning error conditions such as device off-line which formerly required polling of the target.





# Building a Semi-Loosely Coupled Multiprocessor System Based on Network Process Extension

Helen S. Raizen (raizen@s35.prime.com)  
Stephen C. Schwarm (schwarm@vino.dec.com)<sup>1</sup>

Prime Computer, Inc.  
500 Old Connecticut Path  
Framingham, MA 01701

## Abstract

A semi-loosely coupled (SLC) architecture for multiprocessors is proposed in order to achieve high availability and scalable performance. The paper goes on to propose a software architecture to solve the problem of making multiple copies of an operating system look like a single system to users, applications and administrators. The software consists of a Distributed Unix<sup>2</sup> System Call Library (DUSClib) that is linked to applications via SVR4 dynamic linking. Underlying DUSClib is Network Process Extension (NPX<sup>3</sup>), a master/slave (as opposed to client/server) based remote procedure call mechanism that enables fast, simple implementation of distributed systems software. NPX is described in detail and its advantages over client/server mechanisms are discussed. A prototype of an SLC system has been built with a DUSClib over NPX and a relational database management system (Oracle<sup>4</sup>) has been run on the prototype. Preliminary performance results show the possibility of scalable performance, but are inconclusive.

## 1. Motivation for Building a Semi-Loosely Coupled System

The realities of the computer marketplace today require systems that can be expanded at reasonable incremental cost and are based on commodity processors. The tightly coupled symmetric multiprocessor (SMP) system architecture is the most popular computer architecture in today's marketplace for using commodity processor chips to achieve high levels of performance at relatively low cost. But there are some inherent drawbacks to the tightly coupled SMP approach:

- **the problem of availability:** When a problem occurs on a single-user workstation, typically only one employee's work is affected while the workstation reboots or the user locates an idle substitute (for a hardware problem). When a problem occurs on a large tightly-coupled SMP system providing central services and databases for many users, all

---

<sup>1</sup>S.C. Schwarm was at Prime Computer, Inc. at the time that most of this work was done. He is now at Digital Equipment Corporation

<sup>2</sup>Unix is a registered trademark of UNIX System Laboratories, Inc.

<sup>3</sup>NPX is a trademark of Prime Computer, Inc.

<sup>4</sup>Oracle is a registered trademark of Oracle Corp.

users are affected resulting in significant work loss. A common solution to this problem is a fault-tolerant architecture, but this requires the expensive duplication of most hardware components and only addresses hardware-related problems; operating system panics will continue to bring down an entire tightly-coupled system. [21]

- **the problem of scalability:** There are inherent limits on the performance of tightly coupled SMPs. As more processors are added, a performance bottleneck is eventually reached. The bottleneck may be the system bus, some system or application resource that are single-threaded, or the cost of unlimited process migration. [1, 16] Our own unpublished modeling results, as well as those of Jog *et al* [17], show severe performance scaling limitations in the six to eight processor range for an interactive database benchmark. The problems that limit scalability can be addressed with engineering solutions, but not without additional cost in hardware and/or in software development.

The problem of availability can be addressed in a loosely coupled environment, especially if users connect to the systems via Ethernet and file systems are replicated. [12, 29] When one system goes down, a user simply logs into another system in the network and continues working. But loosely coupled (distributed) systems have their own set of drawbacks:

- **the problem of network performance:** The latency of network transmissions and the low reliability of networks both lead to unacceptable response times in loosely coupled systems. In particular, network overhead often requires more processing than the operation that is being done remotely. [6]
- **the problem of distributed locking:** Most database management systems do fine-grained locking using hardware test and set instructions. The overhead of transferring these locking schemes into loosely coupled systems is usually excessive. Though many alternatives have been explored to alleviate this problem, performance of commercial database systems is often unacceptable when the database is distributed. [5, 7]
- **the problem of load balancing:** Even if network latency isn't a problem, tightly coupled systems are regarded to be easier to balance load across processors. With a loosely coupled (networked) system, it is hard to tell where there is idle processing power and it is difficult to shift load around. [18, 22]

Our project seeks to demonstrate that there is a feasible middle ground for multiprocessor systems between the loosely coupled and tightly coupled architectures that will give a close equivalent in performance to a tightly coupled SMP while having the potential for high availability found in a networked environment, and without the cost of fault tolerant systems. Such a system should also be scalable to very large numbers of processors and address the issue of load balancing. In order to reach this middle ground, we propose a semi-loosely coupled (SLC) architecture that would allow us to invoke multiple operating systems (an OS for each processor, or pair of processors) while presenting what looks like a single operating system to users, applications and administrators. By running separate operating system invocations on each of the multiple processors, an operating system or processor failure is contained to that processor; users on other processors continue uninterrupted. While common system hardware (e.g. power supplies, buses, etc.) would still affect the system, it's significantly cheaper to apply redundancy techniques to these areas rather than the entire system as a whole. And hardware reliability has significantly improved over the past decade to the point where software reliability and operations greatly dominate system downtime. [14]

At the hardware level, the SLC architecture requires private memory for each operating system invocation and globally shared memory for applications. It also requires some mechanism for fast reliable message passing. At the software level, time to market, cost considerations, and the

desire for open systems lead us to focus on Unix and to require that a single system view be accomplished with:

- Minimal changes to the Unix operating system. This is required in order to keep the cost of development and the cost of operating system upgrades to a minimum.
- No changes to application software (including data base system software) required to run on the system architecture.

Therefore, a key part of the software involves making operating system functions that currently are restricted to a single copy of Unix (such as shared memory, semaphores, message queues and symbolic links) usable across multiple instances of the operating system.

This paper will focus on the software aspects of building an SLC system. In order to make Unix look like a single system to applications, when in fact there are multiple instances of Unix running on multiple processors, we propose replacing the standard C system call library with a distributed unix system call library (DUSClib) to provide the view of a single operating system to application software. DUSClib must support any Unix system call that might go remote when the parts of an application are actually running on different Unix kernels. At Unix SVR4, a library is a very appealing mechanism for distributing applications across operating systems, because of the introduction of dynamic linking that allows an application to run with a new library without needing to reload the application.

## 2. Related Work

Some early dual processor systems resemble the SLC architecture in some aspects. For example, the SEL 88-type dual processor, designed over 20 years ago, had two processors each with private memory and also an area of shared memory. However, the SEL 88 was limited to 2 processors and did not hide the details of the architecture from applications. [21]

AAMP (Agressive Advantage of Multiple Processors) [3] was developed to address the problem of operating systems and application migration in response to technological changes. While the motivation is different, AAMP involves running multiple operating system invocations on a shared memory multiprocessor. Unlike the SLC system, AAMP works in an environment of heterogeneous operating systems. AAMP took the tack of dedicating portions of main memory to each operating system, but there is no notion of global memory. Also AAMP is not trying to distribute applications across operating systems. AAMP uses a client/server based remote procedure call (RPC) implemented on top of a shared memory, message passing scheme. This leads to client/server type relations amongst the operating systems. While the server can continue executing when a client fails, the reverse is not true and thus the server system becomes a single point of failure which is not desirable for achieving high availability. Our software architecture avoids this problem, partly by using a different type of RPC.

Distributed shared memory (DSM) systems [9, 23, 26] provide shared memory for loosely coupled systems as a software abstraction. The main advantage of DSM systems is that they hide the underlying data passing that occurs between loosely coupled systems under a shared memory abstraction which is easier for application programmers to use. [23] However the data passing between processors that still occurs in DSM systems can cause problems in database management system performance. Most such systems implemented on UNIX use Unix shared memory to implement locking schemes. The frequency of access to database locks will cause thrashing of lock pages between systems. [23] By providing globally accessible memory for Unix shared memory regions, the SLC architecture avoids this problem. Also DSM systems are usually built



on networks that are relatively unreliable and slow. [23] We have proposed fast reliable message passing for the SLC architecture.

MemNet is a hardware implementation of DSM supporting shared memory at the physical address level. [9, 26] It passes data around in 32 byte pieces which improves the problem of thrashing on database locks, but does not entirely solve it. In practice, MemNet has not been scalable in performance [26].

A recurring theme in discussions of DSM is that the performance of various design alternatives depends on the nature of the application. [23, 26] Tam *et al* also discuss the fact that while RPC and message passing mechanisms do not take advantage of locality, they do protect address spaces. The SLC architecture provides global physical memory for implementing Unix shared memory which itself is used by applications where memory sharing is important. Thus applications get direct access to data in a way that is tailored to the application. On the other hand, operating system data is protected from access by other nodes and a form of remote procedure call is used for cases, such as file system access, where it makes more sense to move operations to the data than to move data to the operations.

The problem of distributing applications has often been approached by creating new programming language constructs at the application level that will work on existing (and often heterogeneous) operating systems. DACNOS [13] is a recent example of this approach, while Apollo's NCS<sup>5</sup> [10] is an example from the commercial realm. Others [2, 6, 8, 20, 29] have developed entirely new operating systems, often including new language constructs as well. Usually, the goals of these projects are to create a general environment for implementing distributed applications. Our approach is somewhat different, since we are trying to provide a way of distributing existing applications in order to increase scalability and availability in the environment of a homogeneous operating system. Rather than intervene in application code, or within the operating system, we chose to insert a library layer between the application and the operating system that would hide the fact that distribution is happening from the applications without the cost of extensive operating system modification. A similar approach is taken in the system services layer of ZGL [25], but unlike ZGL, our DUSClib does not involve the use of specialized servers to access remote resources.

### 3. Description of the Software Architecture

Making multiple copies of Unix look like a single system to applications is a challenging software task. For the purposes of the SLC architecture, we can restrict the problem to homogeneous operating systems and underlying hardware. In that context, we can take advantage of dynamic linking in Unix SVR4 and implement an enhanced system call library to extend various system functions across the multiple instances of the operating system. Aside from incorporating dynamic linking, there is nothing terribly unique about this approach. Where our software architecture departs from most others is in the mechanism we chose to use for the remote procedure call upon which we implement DUSClib.

Others who have attempted similar projects, though often for very different reasons, have mostly used a client/server approach to distributing UNIX across the independent instances of the operating system. [3, 28] This is a natural approach to take, given the kinds of remote procedure

---

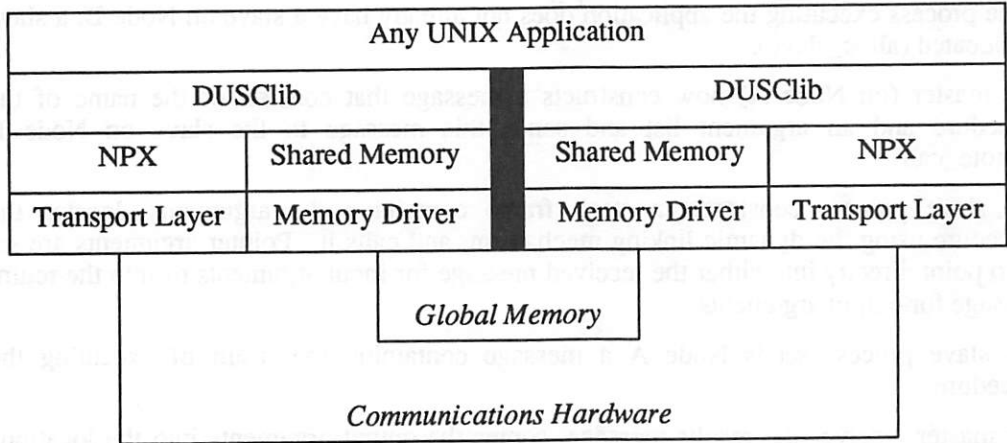
<sup>5</sup>Apollo is a registered trademark of Apollo Computer Inc. NCS is a trademark of Apollo Computer Inc.



call mechanisms that have become standard in the Unix world. We chose to use a shared library to distribute Unix and based our library on a different kind of remote procedure call mechanism. This mechanism, called Network Process Extension (NPX), has been used in Prime's proprietary 50 series<sup>6</sup> for over ten years and has some appealing advantages over the Unix standard remote procedure calls.

In addition to using NPX for most distributed system calls, we had to provide a special mechanism for implementing shared memory. This mechanism had to allow any process on any system to directly access shared memory with little or no performance penalty.

In order to ease porting to different hardware architectures, both the NPX piece and the shared memory piece are divided into hardware/network dependent and independent layers. A diagram of our proposed software architecture is shown in Figure 1.



**Figure 1:** Software Architecture for an SLC System

### 3.1. What is Network Process Extension

Network Process Extension (NPX) is a mechanism for doing remote procedure calls that takes advantage of dynamic linking. While NPX on Prime 50 series computers uses X.25 to go out across the network, the mechanism itself will work over any transport service. The paradigm used for the NPX mechanism is that of master/slave (as opposed to client/server paradigm used in most other remote procedure call mechanisms existing on Unix systems). Client/server implies that the remote location has a specialized process or set of processes (a server) that execute in

<sup>6</sup>Prime is a registered trademark of Prime Computer, Inc. 50 Series is a trademark of Prime Computer, Inc.

order to perform the operations of a particular sub-system for clients. All clients requiring service from a particular server are directed through that process or set of processes. In contrast, the master/slave paradigm implies that the remote location has a surrogate process (slave) that executes in order to perform all the activities of an application process (master) at that remote location. Rather than binding a set of operations to a server, a slave can execute any facilities that can be accessed via dynamic linking, a key enabling technology for NPX. The master/slave paradigm connects an application process to a remote surrogate that can perform all operations needed by that process at that remote location, provided that those operations can be accessed via dynamic linking. The client/server paradigm connects an application process to a series of services available at a remote location, where for each service, there must be a separate communication path established and a separate piece of server code developed. Slaves execute with the identity and rights of the master. Servers execute with the identity and rights of the service. On the other hand, most client/server RPCs support data translation and thus are able to work in heterogeneous environments. NPX currently does not do data translation, though it would be possible to add.

The steps to do a remote procedure call through NPX are as follows:

1. An application calls a procedure on Node A to perform an operation on a resource, F.
2. The procedure code determines that F is remote and located on Node B.
3. If the process executing the application does not already have a slave on Node B, a slave is allocated (`alloc_slave`).
4. The master (on Node A) now constructs a message that consists of the name of the procedure and an argument list and sends this message to the slave on Node B (`remote_call`).
5. The slave process constructs a stack frame containing the arguments, locates the procedure using the dynamic linking mechanism, and calls it. Pointer arguments are set up to point directly into either the received message for input arguments or into the return message for output arguments.
6. The slave process sends Node A a message containing the result of executing the procedure.
7. The master receives the results message, copies the output arguments into the locations specified in the original call and returns to the calling application.

From the point of view of the library implementor, there is a very simple interface to using this mechanism: `alloc_slave`, `remote_call`, `getslaveid`, `dealloc_slave`.

Note that Marionette [24] is a system for parallel distributed programming which uses the term master/slave model differently than we are using it here. In particular, Marionette's slaves are bound at runtime to application specific program binaries, making it much more like a variation of the client/server paradigm as we are using the terms here.

### 3.2. Advantages of NPX over Client/Server

Based on our experience with NPX on the Prime 50 series, there are some significant advantages to using the master/slave paradigm instead of the client/server paradigm, particularly to achieve the goals of the SLC architecture. In particular NPX is superior to other RPC mechanisms in its generality, in easier implementation of distributed services, in performance and

in security.

Tay and Ananda [27] recently surveyed and compared various distinct remote procedure call implementations all of which use the basic client/server paradigm. Each of these RPCs requires the server to be knowledgeable in advance of what it will be asked to execute by clients and most represent requested operations by keys. In contrast, the slave process in NPX contains a **general** mechanism that can execute any procedure by linking to it dynamically based on the name of the procedure. Binding to remotely called procedures is done at run-time and is based on the location of resources rather than services.

Building distributed applications using NPX is easier than using client/server based remote procedure calls, because extensive server code development and use of stub generators become unnecessary. Code for system calls or APIs can be completely symmetrical (executed by either master or slave). Making an API usable in a distributed fashion based on NPX does require building the API into a dynamically linked library. Only a few lines of code need to be added to each routine which may operate on remote resources (assuming that naming and location is handled by a general mechanism). For example, to open a file (once it has been determined to be remote), all that has to be done is to call `alloc_slave`, package up the arguments, call `remote_call`, and unpackage the results. All the procedure specific information is kept within the procedure itself. Once application code has been loaded to use the dynamic libraries, it never has to be recompiled or reloaded, so distribution can be added later without perturbing the application at all.

NPX has advantages because each master has its own slave to achieve remote execution. The one-to-one correspondence of master to slave makes it easy to implement system calls (or other functions) that require waiting. To execute such calls using client/server RPCs requires some form of multi-threaded server [15] which adds to coding complexity and system overhead. If multi-threaded servers are not used, calls that block may throttle performance. Even without blocking calls, multiple clients accessing a single-threaded RPC server remotely may cause a performance bottleneck. These performance problems are inherently absent when using NPX as a remote procedure call mechanism.

NPX has security advantages because, while a server must either change identities to that of each client in turn or be all powerful to access whatever resources each client desires to access, the slave first validates and then assumes the identity of the master and keeps that identity for everything that it does. Note that most of the RPCs surveyed by Tay and Ananda did not contain any security or authentication at all. [27]

### 3.3. Transport Layer

Because slaves are usually allocated and deallocated infrequently relative to the number of remote calls made, it makes sense for NPX to sit on top of a connection-oriented transport service. The actual nature of this layer will be dependent on the hardware configuration. Ideally, NPX would be implemented on top of the Transport Layer Interface (TLI) enabling the calls to go over any communications software/hardware. With adequate support for naming and location, NPX can even support more than one transport mechanism. Note also, that by implementing NPX on a connection-oriented transport service, we achieve at-most-once semantics.

### 3.4. Distributed Unix System Call Library

Using NPX, it is fairly simple to change UNIX system calls to support the illusion of a single system for those applications that are built to dynamically link to the system library (the default at SVR4). We replace the standard system library with one that has been modified to support remote operations. All system calls execute a library interlude which sets up the arguments and then invokes the appropriate kernel function. To convert a given library routine to support remote operations, we simply determine if the resource to be operated upon is remote, use `getslaveid()` and/or `alloc_slave()` to make sure that a slave exists and then perform the appropriate kernel function remotely via `remote_call()`. Dynamic linking is used within the slave to find the system call on the remote node. Note that the library interlude need not be called on the remote node as the slave is guaranteed to be operating on a local resource though in some cases it may be simpler to call through the library interlude.

The difficult part becomes recognizing when a resource is remote and locating the node on which it resides. If the resource is designated by a path name, then the naming and location problem will get resolved by whatever remote file system mechanism is chosen. (This is not to say that the naming and location of remote files is simple, just that it is a higher level problem than we are addressing here.) However, if the resource belongs to Unix such as a file descriptor, an IPC id, or a process id, there must be some mechanism built into the DUSClib to recognize that the resource is remote and locate its host node. This is a non-trivial problem, especially if one chooses to address it with no support from the Unix kernel. One possible solution is to keep translation tables at the library level which convert all identifiers and descriptors from a local namespace to a global namespace. Another possible solution is to encode a node id in the high order bits of the UNIX descriptor or id.

In addition to the naming and location problem, forking presents a particular difficulty when implementing DUSClib via NPX. (This is not a problem that PRIMOS<sup>7</sup> had to confront in the original NPX implementation.) Since both parent and child can continue to execute after a fork and operating system resources such as file descriptors are copied (in essence) from parent to child, it makes sense that when a process forks, all its slaves on remote nodes must fork as well. However, the overhead of this is high, especially if the first thing the child does is an `exec()` or a `close()` on all the remotely accessed file descriptors.

Lastly, signalling is a particularly difficult issue to handle for DUSClib. NPX in PRIMOS has a mechanism for passing "conditions" (similar to signals) from the slave back to the master. However on UNIX, one must also consider the case where a signal must be passed from the master to the slave. For example, consider the common case where an alarm is set, followed by a `semop`. If the semaphore is remote, there needs to be some mechanism for either transferring the alarm() along with the `semop()` or for sending the signal across to the slave, if the alarm goes off while the slave is performing the `semop()`.

### 3.5. UNIX Shared Memory

Making UNIX shared memory support the illusion of a single system is a special case, since the SLC architecture proposes to support shared memory in hardware. For shared memory, changes must be made within the UNIX kernel. The changes required are fairly straight forward. The

---

<sup>7</sup>PRIMOS is a registered trademark of Prime Computer, Inc.



kmem driver must be cloned to provide a global memory driver which enables the mapping of non-local physical addresses. In addition, since UNIX determines the size of virtual memory space by available memory at coldstart (and the individual nodes do not "know" about the shared memory), the kernel's virtual memory space must be expanded. Lastly the shm routines must be modified to allocate, map and deallocate regions from the non-local memory. There may be cases where an application would be better served by allocating shared memory locally, rather than in the global memory area. The only way to accommodate this is to add an additional optional flag for shmget(), IPC\_LOCAL. While this violates the requirements listed for an SLC architecture, the shared memory piece is so fundamental to the architecture that it may be justified. Also, this change only affects performance; applications will function correctly without it.

## 4. Prototype Effort

Since software, rather than hardware, seemed the harder part of bringing an SLC system into existence, the main purpose of the prototype is to demonstrate the feasibility of building an SLC system with a software architecture based on NPX. Also, while the promise of increased availability on an SLC system is a strong motivating factor and the problem of load balancing must be addressed, the question of whether an SLC system would provide scalable performance was the overriding issue that led us into the prototyping effort. Thus the goals of our prototype are:

1. Demonstrate that the software architecture proposed for an SLC UNIX system (based on NPX) can be accomplished in a reasonable amount of time.
2. Demonstrate that ORACLE performance on an SLC UNIX system will scale in an equivalent fashion as on a comparable tightly coupled SMP system.

Because we chose Oracle as the application with which to demonstrate scalable performance and because the object was prototyping, we chose to implement only enough of the software architecture to run Oracle benchmarks. In general we ignored most security issues and we even chose to ignore performance issues that would not directly affect scalability of performance.

### 4.1. Prototype Hardware

The prototype hardware consists of from 1 to 6 CPU boards containing Intel 386 processors and plugged into a Multibus II backplane<sup>8</sup>. Each CPU board has 8MB of local memory. For global memory we have three 4MB Multibus II memory boards. The CPU boards use SCSI buses to connect to disks and tapedrives. The purpose of this hardware is simply to be a substrate on which to develop the software and take performance measurements.

### 4.2. Porting NPX to Unix

To implement NPX on Unix, we simply translated the PRIMOS design into C, using appropriate Unix constructs. We modified the PRIMOS interface slightly to provide some performance improvements and to make it more Unix-like. We left out the internode security

---

<sup>8</sup>Intel is a registered trademark of Intel Corp. 386 and Multibus are trademarks of Intel Corp

mechanisms which are necessary when using NPX in an untrusted network. The slave mechanism in PRIMOS (which preallocated slaves) was simplified to use a slave daemon which forks off slaves as necessary. The part of the slave code that invokes the remote procedure was written in assembler. While PRIMOS uses dynamic linking to bind to each routine that the slave invokes, for the purposes of this prototype we needed to support only one routine, `syscall()`. We did not take the time to implement dynamic linking in the slave code.

In PRIMOS, NPX supports both blocking and non-blocking remote calls; only blocking calls were ported to Unix for the prototype. We determined that we did not need to support forking in slaves in order to run Oracle, so we deferred implementing it. We also chose to ignore the signalling issues for the prototype.

We coded the NPX functions on top of a transport-independent interface which we implemented first on message queues to emulate multiple nodes and allow single-node testing of the software. When we were ready to run on multiple nodes, we recoded this layer to run on top of `mb2lib` which passes messages over Multibus II using the `mb2-tai` driver. We also designed an implementation of the transport layer on top of TLI. It would be fairly straightforward to move the prototype version of NPX to run on TLI. (Moving to the Multibus II interface took about a week.)

### 4.3. Partial Implementation of DUSClib

For the purposes of the prototype, we changed only those library routines necessary to get Oracle running. We chose to implement a distributed file system via NPX, because this was easier than any alternative we considered. In addition to the file system, we made the semaphore functions operate in a distributed fashion. In order to set up Oracle to run in a distributed fashion, we implemented distributed symbolic links.

In order to simplify the NPX port, we chose not to support dynamic linking within NPX for the prototype. Instead, we extended Unix's `syscall` mechanism to support, at the library level, all of the routines that we were making distributed. (Many were already executed via `syscall`).

We were not interested in exploring naming and location issues, so we made a number of simplifying assumptions in this area. We set one node to be the "supernode" and allowed it to be the owner of all semaphores. File descriptors were modified to encode the node id in the high order bits. We modified file system naming to support `<nodeid>` prepended on a pathname to indicate remote location. Thus `/2/` refers to root on node 2.

We decided to run all of the Oracle background processes on one node and believed that this would allow us to ignore signaling and avoid having to implement a global space for process ids. We were wrong on both these issues, but were able to code work-arounds that were good enough to run Oracle.

### 4.4. Global Memory Support

For a real system, the changes to support global memory and use it for Unix shared memory would need to be made within the kernel to protect the memory from unauthorized access. For the prototype, we made most of the changes at the library level because it was simpler and easier. Because local memory was already accessed via Multibus II, few changes were required to access

the global memory. The Unix routine, `mmap(2)` is used to map memory during a `shmat(2)` call. We stubbed `detach (shmdt(2))`. Data structures to allocate and deallocate portions of shared memory are kept in the first page of the shared memory. We relied on the supernode to control certain operations that needed to be synchronized, but most memory operations can be done from any node. The global memory is made permanently resident so we don't have to deal with paging. Lastly, we disabled caching in order to keep the global memory in the prototype coherent. (Note that while this will affect the absolute performance numbers, it should have no effect on the scaling of performance.)

## 5. Performance Testing and Results

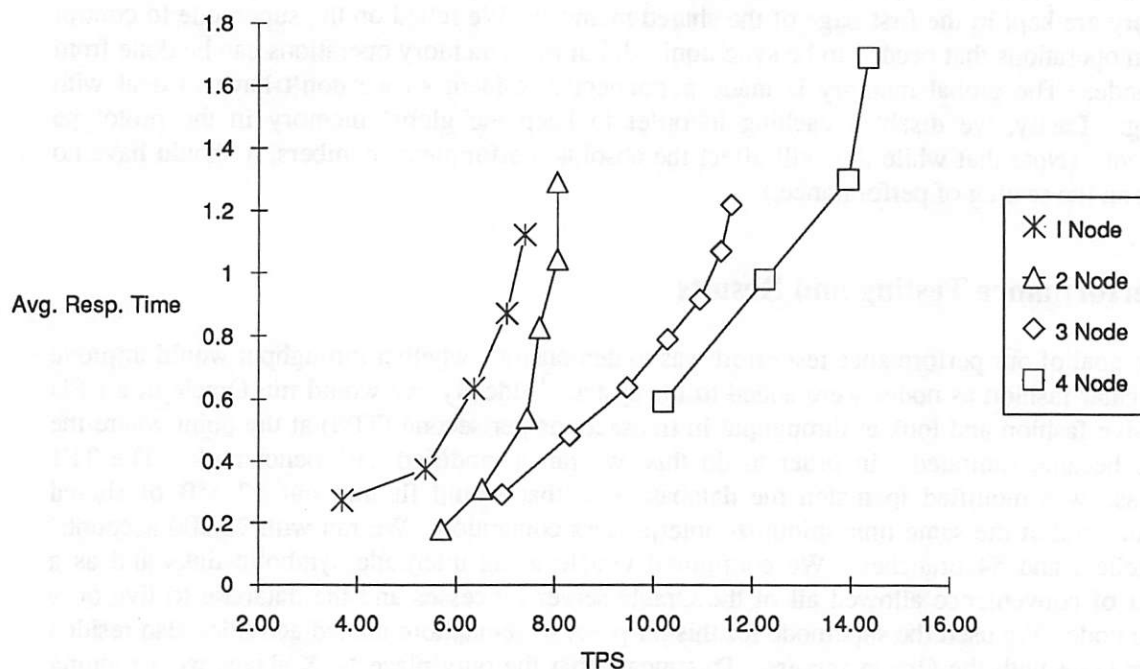
The goal of our performance test effort was to demonstrate whether throughput would improve in a scalar fashion as nodes were added to the system.<sup>9</sup> Ideally, we would run Oracle in a CPU intensive fashion and look at throughput in transactions per second (TPS) at the point where the CPUs became saturated. In order to do this, we ran a modified TP1 benchmark. The TP1 database was modified to match the database size that would fit into our 12 MB of shared memory and at the same time minimize interprocess contention. We ran with 70,200 accounts, 540 tellers and 54 branches. We configured Oracle using internode symbolic links and as a matter of convenience allowed all of the Oracle server processes and the database to live on a single node. We used the supernode for this purpose, so semaphore related activities also resided on the node with the Oracle servers. This meant that the only place NPX slaves were running was on the supernode. With the exception of the single node tests, we ran no user processes on the supernode. Thus single node tests are not directly comparable to the two to six node tests, where the supernode served as something like a database server node. We configured the system this way, because it was easy to do and not necessarily because it was best. We believed that with this configuration, if we could show scaling of two to six nodes, that would adequately demonstrate scalable performance.

Figures 2 and 3 show some preliminary performance results. At this point, we still have some unresolved problems with the performance tests. First of all, the test itself has exhibited a greater degree of variability than we would like to see and this leads us to believe that we have yet to configure the tests in a manner that controls all the test conditions adequately. Secondly, we have been unable to run the tests for five and six nodes with enough users, because we ran out of Multibus II ports. Thirdly, while the system runs to CPU saturation at one and two nodes, it begins to bottleneck on something besides CPU for three or more nodes. For example, a load of four users per node used 77% CPU on a two node system, but only an average of 30% CPU per processor on a six node system. We have yet to determine whether this bottleneck is a structural or tuning problem with Oracle, a problem with the way we configured Oracle, or a flaw in the design of our system.

Figure 2 plots average response time as a function of TPS for one to four nodes. (We did not have enough data points to plot five and six nodes). Note that the two node system shows improved performance over the one node system below saturation, but converges on single node performance as the system saturates. Given the way we configured Oracle for these tests, this is an expected result. While the three node test does not completely saturate CPU, it surpasses the two node performance by 3 to 4 TPS. By the four node curve, the bottleneck is apparent and the

---

<sup>9</sup>We had hoped to compare performance to a tightly coupled system, but unexpectedly lost access to such a system and were unable to run the comparison tests.



**Figure 2:** Average Response Time as a Function of Throughput

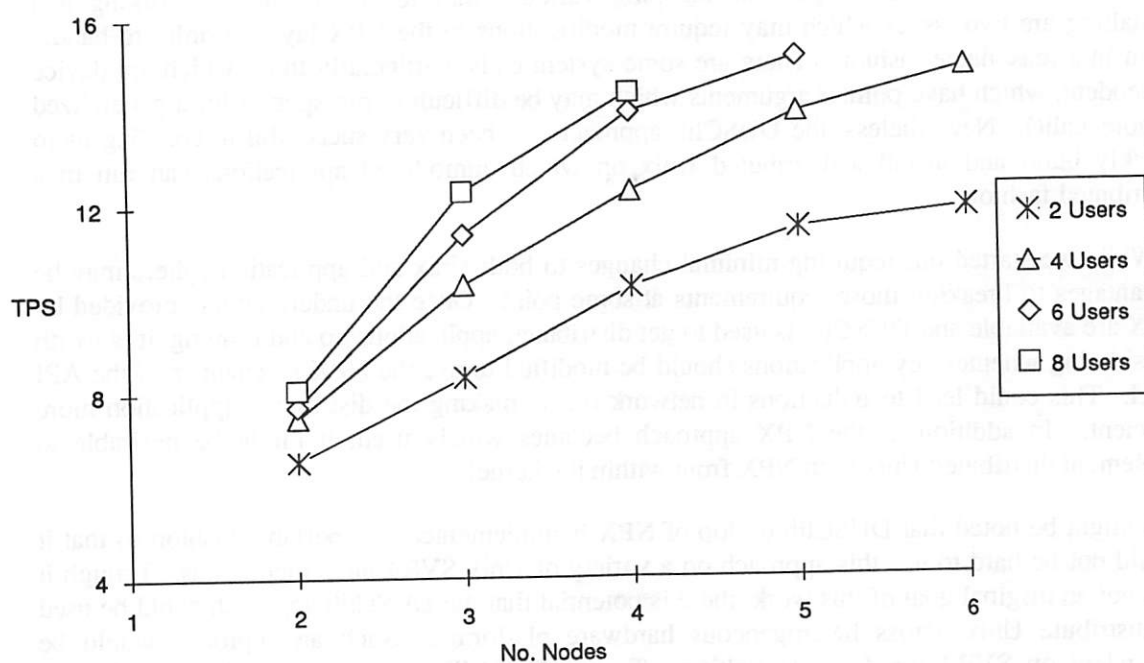
additional CPU adds only 2 to 3 TPS. Without knowing what has caused the bottleneck, we can neither verify or disprove our hypothesis of scalar performance.

Figure 3 plots throughput (in TPS) as a function of number of nodes where each line represents a constant load in users/node. If performance were perfectly scalar, these lines would be straight and at low user load, they are close to straight. This figure suggests a bottleneck which may be related to the total number of users. Such a bottleneck could be due to contention within Oracle, to contention at the Multibus II layer, or to contention caused by our implementation of distributed IPC semaphores.

## 6. Lessons from the Prototype

The prototype demonstrates the ease with which NPX can be ported to a UNIX environment. It further demonstrates that a distributed UNIX system call library can be built using NPX as a base. The portion of DUSClib that we built is installed on each node, dynamically linked to Oracle, and operates in a mostly symmetrical fashion. (Where it is not symmetrical, it is because of simplifications we made to avoid full naming and location.) It supports a distributed file system and distributed use of System V semaphores. Sometimes we even operated through our library without realizing it (as when we used `truss(1)` which uses semaphores and slaves were invoked to handle all the semaphore operations). The library worked so smoothly that we only became aware that we were running in a distributed fashion by accident. This is precisely the quality that a fully functional DUSClib should have, that applications will run and access distributed resources without the application even being aware that this is happening.





**Figure 3:** Throughput as Function of Total Processor Nodes for a Constant Number of Users Per Node (other than the Supernode)

We believe that the prototype shows that the software architecture that we proposed for implementing an SLC system is quite feasible. Building DUSClib on top of NPX verified many of our claims about the advantages of NPX. Emulating basic Unix file I/O on Amoeba took about two weeks of work [20], but we built a distributed file system, modifying a similar set of system calls to go remote on top of NPX in just a couple of days. To make a new subsystem remote requires a small amount of up-front design. Once this is completed, each routine takes a few minutes to implement. Considerably more time is spent writing test programs than implementing the remote routines themselves.

On the other hand, during performance testing we noted a possible disadvantage of the NPX approach when resources are limited. Since each NPX slave used its own communications port, we ran out of ports more quickly than we might have if we had used a client/server architecture with a file server and a semaphore server. Note that neither approach is immune to resource shortages, because as you add different types of servers, or multi-threaded servers, resource shortages can affect the client/server approach as well. A tradeoff will be made, in either case, between performance and flexibility to avoid resource limitations. We believe that NPX still has an advantage over the client/server approach in allowing the autonomous scheduling of each user's work.

The prototype version of DUSClib showed that a dynamically linked library could layer enough distributed features onto Unix SVR4 to run a very complex application, namely Oracle, in a distributed fashion with no modifications to that application code. The fact that the library is dynamically linked saved us countless hours during debugging, because it was much faster to

rebuild and re-install DUSClib than to do the same for Oracle. To fully implement DUSClib, however, there are some difficult problems that we have not yet addressed. In particular, better mechanisms must be developed for mapping various Unix ids to locations. Forking and signalling are two issues which may require modifications to the NPX layer in order to handle them in a reasonable fashion. There are some system calls, particularly those which are device dependent, which have pointer arguments which may be difficult to pre-specify for a generalized `remote_call()`. Nevertheless, the DUSClib approach has been very successful in enabling us to quickly build and install a distributed Unix on which unmodified applications can run in a distributed fashion.

While we started out requiring minimal changes to both Unix and applications, there may be advantages to breaking those requirements at some point. Once the underpinnings provided by NPX are available and DUSClib is used to get distributed applications up and running, it is worth considering whether key applications should be modified to use the NPX mechanism at the API level. This could lead to reductions in network traffic making the distributed application more efficient. In addition, if the NPX approach becomes widely used, it might be desirable to implement distributed Unix with NPX from within the kernel.

It might be noted that DUSClib on top of NPX is implemented in a portable fashion so that it would not be hard to use this approach on a variety of Unix SVR4 implementations. Though it was not an original goal of this work, there is potential that the DUSClib approach could be used to distribute Unix across heterogeneous hardware platforms. Such an approach would be dependent on SVR4 for dynamic linking. To use DUSClib as a generalized mechanism of distributing UNIX, a different approach to Unix shared memory would have to be used. There are several possible ways to address this issue coming out of the work on distributed shared memory [9, 23, 26]. Some solution would also need to be found to the general problem of pointer arguments where the argument pointed to is not clearly sized. Still DUSClib might be a simple and elegant way of distributing UNIX in more general cases than the SLC system.

## 7. Future Work

Further work is needed in performance to determine exactly what is bottlenecking our test at higher numbers of nodes and/or users and whether this bottleneck is inherent in the SLC architecture or can be resolved by better understanding how to tune Oracle for this architecture. In addition, we need to increase the number of Multibus II ports on each node in order to run as many as forty or fifty users on the five and six node systems. It is hoped that when we do this we won't run out of some other resource. To complete the performance work it would be worthwhile to try some other configurations of Oracle, perhaps with multiprocess Oracle servers spread over multiple nodes and with the database files split over multiple nodes as well.

When we embarked on this work, we wanted to demonstrate the feasibility and usefulness of the SLC approach to building more highly available multiprocessor systems. Our work to date has mainly addressed the problem of making multiple invocations of the operating system look like a single system to applications and users. In the future, we intend to complete studies on how this architecture will affect availability. We also need to address the issue of load balancing on an SLC architecture. Approaches to load balancing on loosely coupled systems [4, 11, 19] might be adapted for use on an SLC architecture. Lastly, for the SLC architecture to be successful, we assumed the existence of a fast and reliable mechanism for message passing. Further exploration in this area would be valuable.

## 8. Availability

The NPX and DUSClib software system described in this paper is not currently available for use by others, but any specific requests for it will be considered. Contact Helen Raizen at Prime Computer.

## 9. Acknowledgments

Tom Kincaid, Joe McIlwain and Dave Peterson worked with us to bring the prototype into existence, contributing many valuable ideas along the way. We wish to thank Nancy Leblang, Don Markuson and Barry Wolman for their critical review of earlier drafts and Cheryl Gottlieb for her help with the performance testing. Lastly, we wish to thank Walter A. Jones, Jr. and Margaret Hannemann for providing us the time and resources to build the prototype.

## References

1. T.E. Anderson, E.D. Lazowska and H.M. Levy. "Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors". Proceedings of ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, Berkeley, CA, 1989, pp. 49-60.
2. F. Armand, M. Gien, F. Herrmann and M. Rozier. "Revolution 89 or 'Distributing Unix Brings it Back to its Original Virtues'". Proceedings of the USENIX Workshop on Experiences with Distributed and Multiprocessor Systems, Fort Lauderdale, FL, 1989, pp. 153-174.
3. B. Beck. "AAMP: A Multiprocessor Approach for Operating System and Application Migration". *Operating Systems Review* 24, 2 (April 1990), 41-55.
4. F. Bonomi, P.J. Fleming and P.D. Steinberg. "Distributing Processes in Loosely-Coupled UNIX Multiprocessor Systems". Proceedings of the USENIX Technical Conference, Baltimore, Maryland, Summer, 1989, pp. 61-72.
5. M.J. Carey and M. Livny. "Distributed Concurrency Control Performance: A Study of Algorithms, Distribution, and Replication". Proceedings of the 14th International Conference on Very Large Data Bases, Los Angeles, CA, 1988, pp. 13-25.
6. D.R. Cheriton and W. Zwaenepoel. "The Distributed V Kernel and Its Performance for Diskless Workstations". Proceedings of 9th ACM Symposium on Operating Systems Principles, Oct., 1983, pp. 129-139.
7. B. Ciciani, D.M. Dias, B.R. Iyer and P.S. Yu. "A Hybrid Distributed Centralized System Structure for Transaction Processing". *IEEE Transactions on Software Engineering* 16, 8 (Aug. 1990), 791-806.
8. D.L. Cohn, W.P. Delaney and K.M. Tracey. "ARCADE: A Platform for Heterogeneous Distributed Operating Systems". Proceedings of the USENIX Workshop on Experiences with Distributed and Multiprocessor Systems, Fort Lauderdale, FL, 1989, pp. 373-390.
9. G.S. Delp, A.S. Sethi, and D.J. Farber. "An Analysis of Memnet: An Experiment in High-Speed Shared-Memory Local Networking". Proceedings of ACM SIGCOMM Symposium on Communications Architectures and Protocols, Stanford, CA, 1988, pp. 165-174.

10. T.H. Dineen, P.J. Leach, N.W. Mishkin, J.N. Pato and G.L. Wyant. "The Network Computing Architecture and System: An Environment for Developing Distributed Applications". Proceedings of the USENIX Technical Conference, Phoenix, Arizona, Summer, 1987, pp. 385-398.
11. F. Douglass. "Process Migration in the Sprite Operating System". Tech. Rept. UCB/CSD 87/343, Computer Science Division, University of California -- Berkeley, February, 1987.
12. A. El Abbadi and S. Toueg. "Maintaining Availability in Partitioned Replicated Databases". *ACM Transactions on Database Systems* 14, 2 (June 1989), 264-290.
13. K. Geihs and U. Hollberg. "Retrospective on DACNOS". *Communications of the ACM* 33, 4 (April 1990), 439-448.
14. J. Gray. "A Census of Tandem System Availability Between 1985 and 1990". *IEEE Transactions on Reliability* 39, 4 (Oct. 1990), 409-418.
15. D. Hensgen and R. Finkel. "Dynamic server squads in Yackos". Proceedings of the USENIX Workshop on Experiences with Distributed and Multiprocessor Systems, Fort Lauderdale, FL, 1989, pp. 73-89.
16. M.D. Hill and J.R. Larus. "Cache Considerations for Multiprocessor Programmers". *Communications of the ACM* 33, 8 (August 1990), 97-102.
17. R. Jog, P.L. Vitale and J.R. Callister. "Performance Evaluation of a Commercial Cache-Coherent Shared Memory Multiprocessor". Proceedings of ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, Boulder, CO, 1990, pp. 173-182.
18. K. Kyrimis. "Placement of Processes and Files in Distributed Systems". Tech. Rept. CS-TR-250-90, Princeton University Department of Computer Science, June, 1990.
19. M.J. Litzkow, M. Livny and M.W. Mutka. "Condor -- A Hunter of Idle Workstations". Tech. Rept. 730, Computer Sciences Department, University of Wisconsin -- Madison, December, 1987.
20. S.J. Mullender, G. van Russum, A. S. Tannenbaum, R. van Renesse and H. van Staveren. "Amoeba -- A Distributed Operating System for the 1990s". *Computer* 23, 5 (May 1990), 44-53.
21. O. Serlin. "Fault-Tolerant Systems in Commercial Applications". *Computer* 17, 8 (August 1984), 19-30.
22. N.G. Shivaratri and P. Krueger. "Two Adaptive Location Policies for Global Scheduling Algorithms". Proceedings of IEEE 10th International Conference on Distributed Computer Systems, Paris, France, 1990, pp. 502-509.
23. M. Stumm and S. Zhou. "Algorithms Implementing Distributed Shared Memory". *Computer* 23, 5 (May 1990), 54-64.
24. M. Sullivan and D. Anderson. "Marionette: a System for Parallel Distributed Programming Using a Master/Slave Model". Proceedings of IEEE 9th International Conference on Distributed Computer Systems, Newport Beach, CA, 1989, pp. 181-188.
25. Z. Sun, X. Xue, J. Zhou, P. Yang and X. Xu. "Developing a Heterogenous Distributed Operating System". *Operating Systems Review* 22, 2 (April 1988), 24-31.
26. M.C. Tam, J.M. Smith and D.J. Farber. "A Taxonomy-Based Comparison of Several Distributed Shared Memory Systems". *Operating Systems Review* 24, 3 (June 1990), 40-67.



27. B.H. Tay and A.L. Ananda. "A Survey of Remote Procedure Calls". *Operating Systems Review* 24, 3 (July 1990), 68-79.
28. G.W. Treese. "Berkeley UNIX on 1000 Workstations: Athena Changes to 4.3BSD". Proceedings of the USENIX Technical Conference, Dallas, Texas, Winter, 1988, pp. 175-182.
29. B. Walker, G. Popek, R. English, C. Kline and G. Thiel. "The LOCUS Distributed Operating System". Proceedings of 9th ACM Symposium on Operating Systems Principles, Oct., 1983, pp. 49-70.

1. The first part of the paper discusses the background and motivation for the development of the SEDMS II system. It also describes the architecture of the system and the role of the various components.

2. The second part of the paper describes the implementation of the SEDMS II system. It discusses the various modules and their interactions, as well as the data structures used in the system.

3. The third part of the paper describes the performance of the SEDMS II system. It presents the results of various experiments and compares them with the results of other systems.

4. The fourth part of the paper discusses the conclusions of the study and the future work that needs to be done.

# Implementation and Performance of a Communication Facility for The Raid Distributed Transaction Processing System\*

*Enrique Mafla*

*Bharat Bhargava*

*Department of Computer Sciences*

*Purdue University*

*West Lafayette, IN 47907*

*E-mail: bb@cs.purdue.edu*

## Abstract

We identify the problems that arise in general purpose interprocess communication mechanisms available for the Raid distributed database transaction processing system by conducting a series of experiments. These mechanisms are CPU-intensive and optimized only for remote communication and do not support multicasting. We develop a transaction-oriented communication facility to address these problems. Its performance is limited only by the network device driver and the system call mechanism overheads. Sending a 100-byte message (monocast or multicast) takes 650 microseconds, which includes the overheads. This is approximately 30% of the cost of the corresponding Unix communication facility. Our communication facility demonstrates the feasibility of address spaces for structuring complex distributed transaction processing systems. It employs shared-memory ports, a simple naming scheme, and a transaction-oriented multicasting mechanism. Local and remote communication is through ports. Ports can be accessed directly by the kernel and by user-level processes. The naming scheme used for the application and network levels avoids the use of name-resolution protocols by directly mappings the application-level name space to the network name space. Physical multicasting is used and the need for special protocols to agree on a group address is avoided. Each transaction defines a multicasting group consisting of the set of sites involved. A group's multicasting address is a function of the corresponding transaction identifier and can be independently determined by each member of the group. The new communication facility reduces kernel overhead during transaction processing in Raid by up to 70%.

---

\*This research is supported by NASA and AIRMICS under grant number NAG-1-676, NSF grant IRI-8821398, and AT&T.

# 1 Introduction

Transaction processing systems are large and complex systems. Address spaces can be a useful structuring device for such systems [Ber90]. Under that structuring paradigm, each of the logical components of the system is implemented in a separate address space. The separation of the logical components of the system is enforced by hardware. Many operating systems use address spaces to isolate their functional components [YTR<sup>+</sup>87, RR81, DJA88, Ben87]. Several distributed transaction processing systems have also been developed under this paradigm [LCJS87, BR89b, Duc89].

Expensive interprocess communication facilities results in systems with either poor structure or poor performance [vRvST88, BALL90]. The *small-kernel* or *backplane* paradigm for structuring distributed systems offers several advantages [Ber90, DJA88]. The performance of such a system design demands efficient interprocess communication services [Che84]. If the interprocess communication overhead is high, the designers compromise the structuring of the system to enhance the performance [Ber90, RR81, LMKQ89].

Interprocess communication has been a topic of study in many experimental systems. Lightweight RPC is a high performance cross-address space communication method, which exploits the fact that most communication is local rather than remote [BALL90]. User-level RPC frees the kernel from communication processing [Ber90]. In some systems, the integration of virtual memory, interprocess communication, and file systems have been used to improve their performance [RR81, YTR<sup>+</sup>87]. This is also being investigated to improve the communication support for distributed transaction processing systems. In order to reduce communication overhead, Argus uses threads and runs on a modified Unix<sup>1</sup> kernel [LCJS87]. In Camelot, threads have been used to improve its throughput [Duc89].

In this paper, we study the communication problem in the context of the Raid system (a robust and adaptable distributed database system for transaction processing [BR89b]). We use the Raid System as an example since Raid servers represent functions (such as concurrency, atomicity, replication) that are widely recognized as the logical components of a distributed transaction processing system. It is a research vehicle and has been developed on Sun<sup>2</sup> workstations under the Unix operating system. Raid is structured as a server-based system to provide the infrastructure for adaptability [BR89a]. Each major logical component of Raid is implemented as a server, which is a process in a separate address space. Servers interact with other processes only through a high-level communication subsystem, which has been presented in [BMR90]. Servers can migrate to different computing nodes to increase reliability, load balancing, adaptability, and availability. Currently, there are six major subsystems in Raid: User Interface (UI), Action Driver (AD), Access Manager (AM), Atomicity Controller (AC), Concurrency Controller (CC), and Replication Controller (RC).

## 1.1 Objectives of our Research

Our goal is to develop an efficient communication support for distributed transaction processing systems in conventional architectures. By conventional architectures, we mean virtual-memory, single-processor machines with no special hardware support for interprocess communication<sup>3</sup>. Our efficiency measure is the ratio between the CPU time spent on com-

<sup>1</sup>Unix is a trademark of AT&T Bell Laboratories.

<sup>2</sup>Sun is a trademark of Sun Microsystems, Inc.

<sup>3</sup>Some main frame computers have hardware assistance for IPC. More than one address space can be accessed at the same time.



munication and the CPU time spent while processing transaction management algorithms for concurrency and reliability [Bha87]. This measure of efficiency is especially appropriate for local area networks, where network delay is insignificant compared to message processing time.

Address spaces can provide a natural platform for the support of concurrency, reliability, and adaptability. The complete system (including kernel, operating system services, and applications) is decomposed into smaller and simpler modules. The modules are self-contained and interact with each other via well-defined interfaces. First, concurrency benefits because processes are the schedulable units of the system. An input-output interaction will not block the whole system, but only the module that handles that interaction (e.g. the disk manager, or the communication manager). The other modules of the system can still run concurrently. Second, reliability increases because of the hardware-enforced failure isolation of the logical modules of the system. Finally, it is easier to implement short and long term adaptability [BR89a].

In this paper, we identify the principles for efficient communication for transaction processing in the Raid distributed database system. We plan to measure message processing costs and communication overhead in Raid. Next, we design and implement a transaction processing oriented communication mechanism that addresses these issues. Finally, we compare the performance of the transaction processing system running on both a conventional and the new communication subsystems. We contrast the two options based on the ratio: communication-time/processing-time. Communication time is the CPU time overhead introduced by communication. Processing time is the CPU time dedicated to process transaction processing functions: concurrency control, atomicity control, replication control, etc..

## 1.2 Related Research and Relevance to Raid Transaction Processing

Many communication facilities were originally developed for wide area networks where network delays are considerable, and for applications such as remote terminal access and file transfer [Che86]. To support distributed systems, several special-purpose communication facilities have been proposed. We now briefly describe some of the most relevant communication observations and paradigms that have been developed to support distributed systems.

**IPC in Distributed Systems.** Accent is an example of a communication-oriented system [RR81], which integrates virtual memory, file management, and IPC to improve performance. Communication ports are capability-based and software interrupts can be used to receive messages asynchronously. IPC in Mach, a successor of the Accent, takes advantage of page mapping virtual memory mechanism and handoff scheduling [YTR<sup>+</sup>87, JC89]. A distributed transaction system Camelot has been implemented on top of it [STP<sup>+</sup>87], but experiments detected that the load on the operating system is considerable higher than the load on the any part of Camelot [Duc89]. To quote the authors of those experiments: "this is an unavoidable consequence of implementing the transaction manager as a service reachable only via interprocess communication" [Duc89]. VMTP (Versatile Message Transaction Protocol) is a transport level protocol intended to support the intra-system model of distributed processing [Che86]. The complexity of the design and implementation in transaction processing systems can be reduced by using the VMTP model. DUNE's *service request* model extends the remote procedure call model, to allow client-server interaction in

the middle of the calls [PA89]. Dynamic binding of the amount, source, and destination of data involved in the call is also possible.

The performance of server-based transaction processing systems like Raid is also affected by the efficiency of the underlying interprocess communication subsystem. Server-based transaction processing systems are communication-intensive [Maf90]. The integration of the file system, virtual memory mechanism, and interprocess communication facility can have a positive effect on the performance of database transaction systems, where considerable amounts of data have to be passed among servers. The control flow in transaction processing can be determined in advance since messages follow well defined paths between servers. Because of this property, a transaction processing system can take advantage of handoff scheduling. The intra-system model of communication introduced by VMTP can reduce the complexity of the design and implementation in transaction processing systems. Specialized multicasting support is needed for replication, commitment, and recovery control in transaction processing. Sophisticated RPC services like those supported in Dune can be useful for the implementation of object in transaction processing systems.

**Local IPC.** Lightweight RPC and user-level RPC are purposed in [Ber90] to achieve high performance cross-address space communication. In [Che84], the advantages of interprocess communication for structuring operating systems have been discussed; the processor's registers were purposed to be used as a communication channel between processes to avoid copying and context switching overhead, with the help of special scheduling policies (the receiving process has to run immediately after the sending process relinquishes the processor). Shared memory and user-level synchronization have been used in a Firefly multiprocessor in order to increase interprocessor communication performance [SB90].

Like other distributed systems, Raid has need for intensive local communication activity [Ber90, Maf90]. For a transaction with six-operation in a five-site distributed database system, 90% of the communication activity has been found to be local [Maf90]. The user-level remote procedure call and the communication facilities implemented in the Firefly multiprocessor are attractive for supporting efficient local interprocess communication in Raid. Those facilities are based on shared-memory, which can simplify and speed up the transport of high-level, typed messages with pointers among servers.

**Communication Protocols and Local Area Network.** *Virtual protocols* and *layered protocols* are used in the *x*-kernel to implement general-purpose yet efficient remote procedure call protocols [HPAO89]. They can demultiplex messages to appropriate lower-level protocols, bypassing some unnecessary layers. The need for specialized communication protocols for local area networks has been discussed in [Svo86]. The overhead of standard protocols cancels the high communication speed offered by modern local area network technology. Several experiments have shown that communication in local area networks is CPU intensive [BMR87, LZCZ86]. Application-oriented protocols provide opportunities for further optimization. Efficient streamlined protocols for high-speed bulk-data transfer have been implemented and used in local area networks [CLZ87, Zwa85]. Reliable multicasting schemes for local area networks make use of their technology to optimize resource utilization and communication delay [KTHB89, VM90].

**Communication Support for Distributed Transaction Processing.** The functions that a communication subsystem should provide in order to support a distributed transac-

tion processing systems have been identified in the Camelot project [Spe86]. RPC-based session services have been proposed to support the interaction among data servers and applications. Besides the synchronous RPC with *at-most-once* semantics, other forms of RPC such as asynchronous RPC and multicasting RPC are useful. Highly specialized datagram-based communication facilities can be used to satisfy the performance demands of data servers and applications. The communication subsystem can use its knowledge about the nature of the transaction system to improve its services. Specialized communication facilities have also been used to improve the design and implementation of distributed database management systems [Cha84, BJ87, MSM89, KTHB89].

In the next section, we evaluate experimentally and study the problems with conventional interprocess communications facilities to support distributed transaction processing in Raid. Based on those studies, we designed and implemented a new communication mechanism to support distributed transaction processing systems. Section 3 describes this new communication mechanism. Section 3.3 presents performance figures for the new communication mechanism. In that section, we also test the impact of the new communication subsystem on the performance of the Raid distributed database system. Finally, we present conclusions and suggestions for further work in section 4.

## 2 Problems with Conventional Communication Schemes

Despite the advances made, few distributed operating systems are currently in commercial use [vRvST88]. This is due to an unacceptable communication overhead in these systems [Duc89, Che84]. In order to identify the problems with present interprocess communication facilities to support distributed transaction systems structured around address spaces, we conducted a series of experiments on the performance of the present Raid communication subsystem [BMR87, BMR90, MB90].

The experiments are conducted in our Raid laboratory, which is equipped with five Sun 3/50s and four SPARCstations connected to a 10 Mbps Ethernet. The Raid Ethernet is shared with other departmental machines and has connections to other local area networks and to the Internet. All Raid machines have local disks and are also served by departmental file servers. They run the SunOS 4.0 operating system. For our experiments, we used only the Sun 3/50s. One of the workstations is configured with a special microsecond resolution clock, which is used to measure elapsed times<sup>4</sup>.

Our experimental work focuses in the following areas: general purpose interprocess communication facilities, local communication, multicasting, and impact of interprocess communication on distributed transaction processing performance.

We measured the overhead introduced by the layers of the socket-based interprocess communication model for datagram communication (UDP). These layers include the system call mechanism, the socket abstraction, communication protocols (UDP, IP, and Ethernet), and interrupt processing. We found that the most expensive layer was the socket abstraction, which included copying the message between user and kernel spaces. Starting the physical device required approximately 20% of the total send time. Processing of the three communication protocols incidental to IPC (UDP, IP, Ethernet) represented less than 30% of the total time (except for packets that required assembling on the receive side).

<sup>4</sup>This timer board was developed by Peter Danzig and Steve Melvin. It uses the timer chip AM9513A from Advanced Micro Devices, Inc. The timer has a resolution of up to four ticks per microsecond. The overhead to read a timestamp is approximately 20  $\mu$ s.



We also measured the round-trip performance of several local interprocess communication facilities available on SunOS. We investigated several mechanisms including two-message queues, one-message queue, named pipes, shared memory with semaphore, and UDP sockets in both the Internet and Unix domains. Our experiment demonstrates the benefits of special-purpose methods for local IPC. All methods have a constant overhead (null message times) and a variable overhead proportional to the message length.

We studied several approaches to provide multicasting facility for distributed transaction processing. The first two alternatives are based on the SE<sup>5</sup> protocol. The user-level SEmulticast utility is implemented on top of the SE device driver, which provides point-to-point Ethernet communication. The kernel-level SE multicast utility uses the *multiSE* device driver. Finally, we experimented with physical multicasting. The results showed that the time spent on both user-level and kernel-level multicasting are proportional to the number of members in the multicast group, while the kernel-level multicasting always takes constant time. Although physical multicasting minimizes bandwidth, it demands that the multicast address be known to all members of the group, which can incur extra messages.

We observed that the impact of the interprocess communication on the performance of the system is significant from one of our experiments<sup>6</sup>, [Maf90, MB90]. Transactions need a large number of messages and messages are expensive to process. Figure 1 gives a

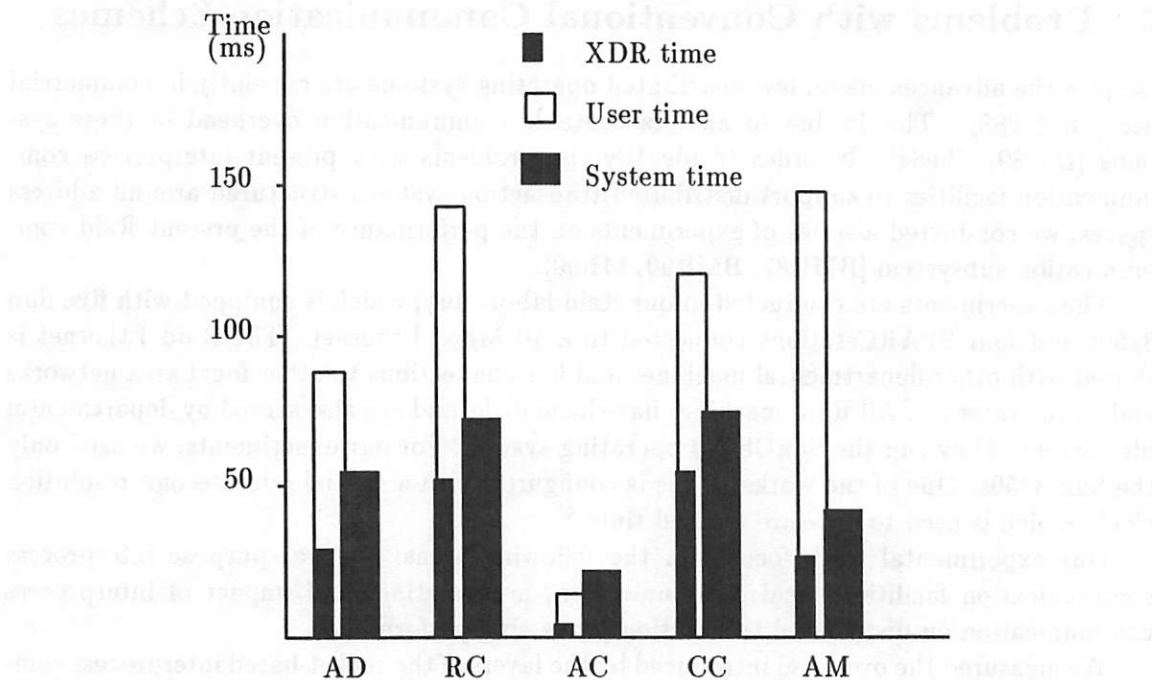


Figure 1: Raid servers' times (in milliseconds).

graphical representation of XDR<sup>7</sup>, user, and system times for Raid servers, for an average transaction. Raid employs XDR to convert complex messages into and from simple buffers before sending them out and after receiving them. User and system times as well as XDR

<sup>5</sup>SE (Simple Ethernet) is a suite of protocols that provide low-level access to the ethernet [BMR87].

<sup>6</sup>We use the DebitCredit benchmark [A<sup>+</sup>85] for our experiments. Known as TP1 or ET1, it is a simple yet realistic transaction processing benchmark, which uses a small banking database consisting of three relations and 100 bytes long tuples.

<sup>7</sup>XDR is a standard for external data representation proposed by Sun Microsystems, Inc.



times are given in seconds. User time is the time spent by the CPU processing user level code. System time is the time the CPU spends executing kernel code. XDR time is the time spent by outgoing and incoming messages in XDR routines.

Details on the experimental setup, procedures, and analysis can be found in [BMR87, BMR90, Maf90, MB90]. Here we summarize the lessons and observations based on experimental studies by listing the following problems with currently used schemes for communication software.

- General purpose communication facilities are top-heavy [BMR87, MB90]. To support generality, they use complex and expensive interprocess communication abstractions and mechanisms. Although these abstractions and mechanisms are useful to support a variety of applications and users, they impose unnecessary overheads during transaction processing. For example, the socket interprocess communication abstraction is a powerful platform for the implementation of diverse communication paradigms [LMKQ89]. The `mbuf`<sup>8</sup> memory management mechanism offers increased flexibility for the implementation of different communication protocols [LMKQ89]. However, efficiency-critical, communication-intensive applications like transaction processing, can not afford the overheads imposed by general purpose communication facilities. In addition, messages have to go through all layers of the communication subsystem even if some of them are irrelevant to the processing of specific messages.
- Transaction processing systems are communication intensive. In a well-decomposed system, most of the communication is local rather than remote [BALL90]. However, most conventional interprocess communication services are designed with the remote case in mind. The local case is handled as a particular instance of the remote case. As a result of this, the operating system kernel becomes the bottleneck of the system, because of both the high message traffic and the high cost to process messages [MB90]. By measuring the performance of several local IPC facilities available on SunOS, we saw that communication facilities that are specialized for the local case are simpler and more efficient. Some of these alternatives, however, are difficult to integrate with the remote communication facility [MB90].
- Some operating systems do not provide enough communication support for distributed transaction processing. The transaction processing system implementor has to supply those services. It is desirable to define high-level interfaces between the modules of complex system [BFH<sup>+</sup>90, RR81]. For communication, modules use typed messages rather than simple buffers of bytes supported by the operating system. To send a message, it has to be marshaled into kernel buffers. The receiving side has to perform the inverse operation. Those are very costly operations (see figure 1).
- General purpose multicasting mechanisms require group initialization and maintenance [CD85]. Multicasting groups that are typical in distributed transaction processing are both dynamic and short lived. In this case, the overhead of group initialization can obliterate the performance advantages of multicasting. Our experiment shows that multicast simulation is CPU and network intensive, and the simulation of multicast inside the kernel reduces CPU overhead [MB90].

<sup>8</sup>`mbuf` is the unit of memory allocation used in the Unix communication subsystem.

- Name resolution can become an expensive and complicated process. In general, we can have three different name spaces: application name space, interprocess communication name space, and network name space. If the name spaces are not related in a simple way, name resolution protocols have to be used for mapping names from one space into another. For instance, the Raid system uses a special protocol to map Raid names into interprocess communication addresses (UDP/IP addresses). These addresses have to be mapped into network addresses (e.g. Ethernet addresses) via a second address resolution protocol.

In the next section, we present the details of our design and implementation of a distributed transaction oriented communication facility. It emphasizes light overhead, simple naming, and transaction-oriented multicast support.

### 3 Implementation of a Transaction-Oriented Communication Facility

We have implemented a communication facility that addresses the problems identified in the previous section. The design of this new communication facility makes use of the insight gained from our experimental work and of ideas proposed in [Spe86, Svo86, Che86, BALL90].

#### 3.1 Design Principles

The design of the new communication facility for distributed transaction processing is based on the following ideas and paradigms learned from experimental studies:

1. Avoid the use of top-heavy communication protocols and abstractions. The experiments [MB90] show that the socket abstraction and general purpose communication protocols unnecessarily increase communication delay. This is true for both local and remote communication.
2. Reduce kernel interaction. Transaction processing is kernel intensive [Duc89]. Our measurements show not only the large number of system calls necessary to process a transaction, but also the large amount of time servers spend at the kernel level. Kernel operations can become the bottleneck, especially in a multiprocessor environment. Efficient communication facilities will reduce system time and context switching activity (more messages processed during a time slice).
3. Minimize the number of times a message has to be copied. This is especially important for local IPC because of the intense local communication activity in a highly-structured system like Raid. Shared memory can be used for that purpose, especially for intra-machine communication.
4. Use a simple IPC memory management mechanism. Most of inter-server communication consists of short and simple control messages. Although the mbuf approach in Unix is very flexible, it has negative effects on IPC performance [MB90].
5. Use the same mechanism for both remote and local communication. This ensures the flexibility and reconfigurability of the transaction processing system. Server will be location independent, We will, however, optimize for the local case.

6. Exploit the nature of a transaction processing system in the design of its underlying communication subsystem. For a LAN, a straightforward correspondence between logical and physical communication addresses can be established. We do not need special protocols to map between logical and physical addresses any more. Group (multicasting) addresses used during commitment time can be determined as a function of the unique transaction ID. This eliminates the need for extra messages to set up group addresses.

To minimize copying, the new communication facility uses shared memory between the kernel and user-level processes. Communication ports are uniquely identified based on the Raid addresses of the corresponding servers. The site number maps to the host address and the triplet (Raid instance number, server type, server instance) maps to a port within the given host.

For multicasting, we use the fact that multicasting groups are formed by servers of the same type. For instance in Raid, we have two types of multicasting groups. One type of multicasting group is used for replication control. Groups of that type can be formed by RC servers only. The other type supports atomicity control and can include only AC servers. The difference between monocast and multicast addresses is in the second component of the addresses. For monocast, we use site numbers, while for multicast, we use the transaction identifiers. A transaction id uniquely determines the members of the multicasting group for a given transaction.

### 3.2 Implementation Details

Our first implementation of the new communication facility is for local area network environments. It is based on shared memory (between processes and the kernel), a simple naming mechanism, and a transaction-oriented multicasting scheme. In this subsection, we discuss ports, which are the basic communication abstraction, the naming and multicasting schemes, and the communication primitives of our new communication facility.

**Ports.** Processes communicate through ports. These ports reside in a memory segment shared by the process and kernel address spaces. Thus, data can be communicated between the kernel and the process without copying. This reduces the amount of copying by 50% compared to other kernel-based IPC methods. The shared memory segment contains a transmission buffer and a set of receiving buffers. The number and length of these buffers are specified by a process at the time it opens a port. The receiving buffers form a circular queue, which is coherently managed by the kernel and the process according to the conventional producer/consumer paradigm. Associated with the transmission buffer and each of the receive buffers there is an integer, which specifies the actual length of the message. In addition, there is a counter for the number of active messages (messages that have arrived, but have not been processed by the server yet). Figure 2 shows the structure of a communication port.

**Naming.** Within a given node, ports are uniquely identified by the triplet (Raid instance number, server type, server instance). The other component of a Raid address, site number or transaction id determines the address of the physical node for monocast or the addresses of the group of nodes for multicast respectively. In the case of Ethernet, we use only multicasting addresses for link level communication. Site numbers or transaction id's are

trmlen	Transmission Buffer	Active Buffers
len1	Receive Buffer 1	
len2	Receive Buffer 2	
len3	Receive Buffer 3	
	.....	
lenN	Receive Buffer N	

Figure 2: Structure of a Communication Port

used to build multicasting addresses by copying them into the four more significant bytes of the Ethernet address (the first bit of a multicast address has to be one).

**Multicasting.** During transaction processing, physical multicasting is used in the following way. While processing data requests for a given transaction, each participant site sets a multicasting address using the transaction ID as its four more significant bytes. When commitment time arrives, the coordinator uses that address to multicast messages to all participant sites. This approach takes full advantage of physical multicast, without incurring the overhead of other multicasting methods. Currently, multicast addresses are added/deleted by Raid servers. The RC adds a new multicasting address for a transaction, when it receives the first operation request for that transaction. In normal conditions, the AC deletes the multicasting address once the transaction is committed or aborted. In the presence of failures, the CC does this job as part of its cleanup procedure. In the future, we plan to manage the multicasting addresses in the communication subsystem. This will improve performance and transparency in Raid.

**Communication primitives.** System calls are provided to open and close a port, to send a message and to add or delete a multicasting address. There is no need for an explicit receive system call. If idle, a receiving process must wait for a signal (and the corresponding message) to arrive.

To send a message, a process writes it into the transmission buffer and passes control to the kernel. If the message is local, it is copied into a receiving buffer of the target port and the owner of the port is signaled<sup>9</sup>. We use the Unix SIGIO signal for this purpose. Otherwise, one of the existing network device drivers is used to send the message to its destination. The destination address is constructed as described above and the message is enqueued into the device's output queue.

When a message arrives over the network, it is demultiplexed to its corresponding port. Again, a signal alerts the receiving process about the incoming message. All this is done at interrupt time and there is no need to schedule further software interrupts.

<sup>9</sup>The process ID of the process that owns a port is stored in the port's data structure.



### 3.3 Performance Evaluation

We conducted two experiments to evaluate the performance of the new communication facility. First, we studied the performance of the basic communication mechanism at the system call level. Second, we tested the impact of the new communication subsystem on the overall performance of Raid. Here, we separately studied the contribution of the new communication subsystem to the local and remote interprocess communication activity in Raid.

#### 3.3.1 Communication Primitives

To evaluate the performance of the basic communication primitives, we measured local and remote round trip times. Figure 3 presents the results of those measurements. For

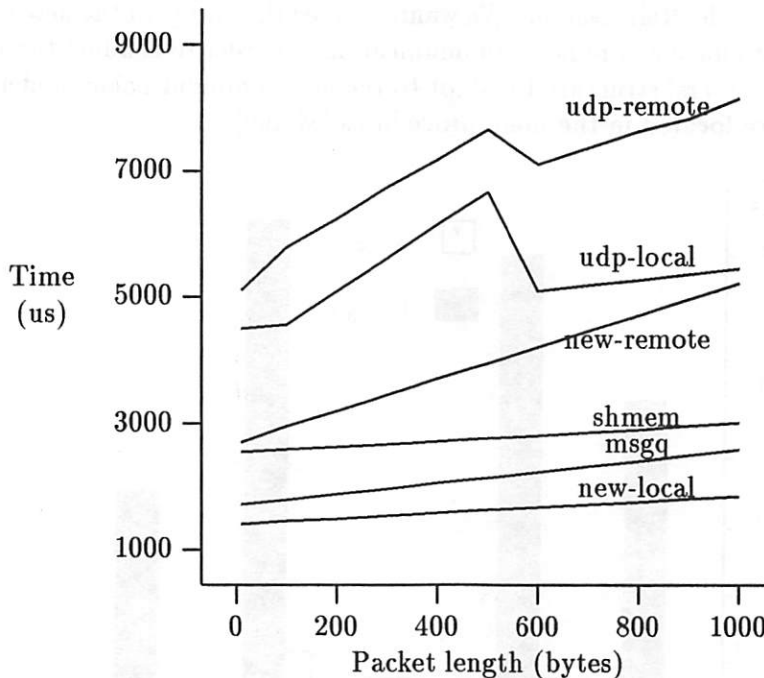


Figure 3: Round trip times (in  $\mu s$ )

comparison purposes, we have added the corresponding times for the SunOS socket-based IPC mechanism using UDP/IP and for two local IPC methods: message passing, and shared memory with message copying.

Both socket-based IPC and the new communication facility provide the same functionality in a LAN environment. Our protocol is extremely lightweight. In the local case, most of the round-trip time is context switching overhead. We measured 1.8 ms for a *round trip of context switches*, representing the context switches necessary to transmit and receive a message. (A process signals another sleeping process and goes to sleep waiting for the same signal from the recently awakened process). We obtained better times for local round trips of messages, because of optimizations done in signaling the receiving process. In the remote case, the network device driver overhead equally affects both methods. This overhead is significant. Despite this fact, the new communication facility achieves improvements of up to 50%. For multicasting, the performance advantages of the new communication facility become even more significant. Sending time does not depend on the number of destinations.

On the other hand, multicasting time for the socket IPC method will grow linearly with the number of destinations.

Socket-based IPC does not optimize for the local case. Local round trips cost almost as much as remote ones (68–88%). In the new communication subsystem, local round trip times are only 35–50% of the corresponding remote round trips.

As in the case of shared memory and message passing, the variable component of communication delay is proportional only to the message length. For the local case, the proportionality constant is the same as that of shared memory, which is the best we can hope for in a kernel-based inter-address space interaction.

### 3.3.2 Impact on Performance of Transaction Processing in Raid

We carried out two experiments to test the impact of the new communication facilities on the performance of the Raid system. We wanted to see the effects of the new communication subsystem on both local and remote communication. In order to conduct these experiments, we modified the servers' structure to adapt to the new communication model. The changes are minor and are located in the main procedures [Maf90].

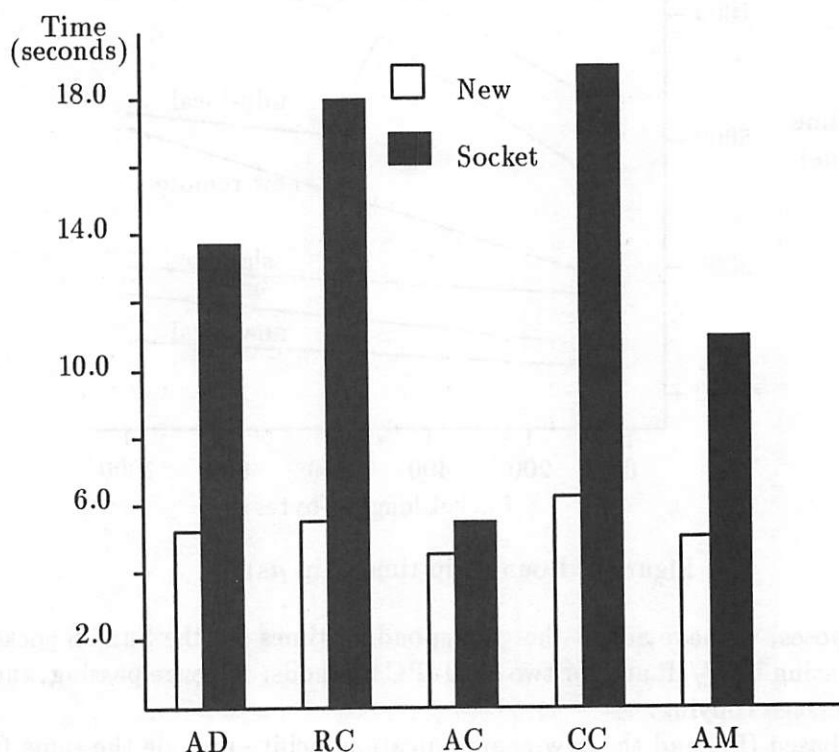


Figure 4: System Time for Raid servers (sec)

For these experiments, we use the same benchmark as in the previous section. We run the benchmark on a single-site and a five-site DebitCredit databases. For the five-site database, we use the ROWA (Read Once Write All) replication method. This means that remote communication is limited to only the AC server. In addition, the benchmark contains 115 transactions that have write operations. Only those transactions need to involve remote sites in the two-phase commit protocol. For details of the experiments, please refer to [Maf90, MB90].

As we discussed in section 2, most of the system time is caused by communication activities. The use of the new communication mechanism allows a 62% reduction of system time for the whole Raid system. Figure 4 shows the savings in system time provided by the new communication facility for the single site case. Savings in user time are less significant [Maf90]. We also noticed a 10% reduction in context switching activities when the new communication library was used. This can be explained by the increased number of messages that can be processed during each time slice.

## 4 Conclusions and Experiences

We identified the communication services that are necessary to efficiently support a well-structured transaction processing system. Separate address spaces can be used to structure a complex transaction processing system. This can provide a natural framework to support high reliability, adaptability, and concurrency. However, that structuring approach increases cross-address space communication activity in the system. When using conventional kernel-based communication channels, the result is a communication-intensive system. Not only is the number of messages high (about 180 messages for a five-operation transaction), but also messages are expensive to process. High interaction among servers also triggers costly context switching activity in the system. Increasing availability through distribution and replication of data demands specialized remote communication. In particular, we need special purpose multicasting mechanisms. The lack of such facilities further degrades the performance of the system.

We have shown that efficient interprocess communication can make the use of address spaces a practical solution for the structuring requirements of complex distributed transaction processing system. In local area networks, the main concern is the performance of the local cross-address space communication mechanism. For a typical transaction, about 90% of the communication activity is local. This fact has been observed for other distributed systems [Ber90]. Conventional interprocess communication facilities are optimized for the remote case [BALL90]. We addressed this problem in the design of our new communication facility. Our first prototype provides a streamlined interprocess communication service. It uses shared memory between the kernel and the server processes. This alleviates in part the demands imposed on the kernel. The use of our prototype in Raid results in a reduction of 60% - 70% of system time. Context switching activity also diminishes because more messages can be processed during the same time slice. The new communication model has a straightforward mapping between server addresses and network addresses. It also exploits the semantics of transaction processing to provide an efficient multicasting support. These two mechanisms have an effect on both system and user times. First, there is no need for explicit address translation processing. Second, the support of physical multicasting eliminates the need to simulate it at the user level.

Communication processing constitutes a significant part of user-level time. At the highest level, servers interact with each other through complex data structures. While the new communication subsystem provides efficient low level, buffer-to-buffer communication, it still does not take into account the high-level communication demands of a transaction processing system. Formatting data structures into buffers can become the major bottleneck of the system. We measured that XDR processing represents approximately 1/3 of user-level time. In the future, we want to explore new local communication paradigms to attack this problem. Shared memory among server processes will avoid the multiple en-

coding/decoding of Raid messages, reducing not only user-level time but also system time. In addition, the kernel will have to do less message processing. The lightweight remote procedure call paradigm in [Ber90] uses shared memory and a special process management mechanism to provide efficient cross address space communication. Communication under that paradigm is kernel-based. The kernel still remains as a potential bottleneck. The user level remote procedure call model in [Ber90] takes the kernel completely out of a message's way. Communication and process scheduling are moved from the kernel into user-level libraries. Not only will it improve the performance of the system (reduced context switching activity), but also and most importantly, will eliminate the kernel as the bottleneck of the system.

Scheduling policies in conventional operating systems do not consider high level relationships that may exist among a group of processes. Optimization of response time or throughput at the operating system level is the main driving force in those scheduling policies. That optimization may not be reflected at higher levels. In other words, conflicts may exist between the optimization criteria at the operating system and application levels. In transaction processing systems, we are interested in response time and throughput not for individual processes but for the whole system of processes. This can be achieved by introducing the concept of a system of processes as a new operating system abstraction. Scheduling can be done at two levels. At the higher level, the kernel would schedule systems of processes as atomic entities. Internally, scheduling could be done based on internal requirements of each system. In particular, it could be based on its communication patterns. In Raid, the concurrency controller is CPU intensive and after some time it has its scheduling priority decreased. This forces CC to give up the CPU after processing only one message, even though its time slice has not expired yet.

Database applications demand the transfer of large amount of data. We believe that efficient bulk data transport services can be provided by exploiting the semantics of transaction processing systems. We need not only efficient file system support but also a closer collaboration between file, network, and communication subsystems. Remote requests for data can be handled within the kernel. This will avoid both kernel-user level interaction and multiple copying and formatting of data. Incremental remote access of data can be also supported as a result of that collaboration [PA89].

Most of our work has been based on local area networks. Wide area network transaction processing systems increase the demands on efficient remote communication. Internetwork-based systems will require more complex communication support. The multicasting scheme of our prototype cannot be used on those cases. Naming and addressing become more elaborate because of the presence of different network technologies. Finally, if the system consists of a large number of nodes, we will need to look for alternative control flows for transaction processing. One of the main objectives of those control flows has to be the reduction of the number of remote messages needed for transaction processing.

New attempts are being made to address these new problems in our Raid laboratory. An adaptable communication facility for distributed transaction processing is under development. In the new facility a more effective mechanism has been developed to transmit complex objects, and new scheduling policies are employed to reflect the high-level dataflow requirements. The new experiment result shows that shared memory between user processes plus user-level scheduling can eliminate the set up time that is needed to transmit complex data messages (about 30% as identified above). The generality can be ensured by multi-channel adaptable approach [BZ91].



## References

- [A<sup>+</sup>85] Anon et al. A measure of transaction processing power. *Datamation*, 31(7):112–118, April 1985.
- [BALL90] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, February 1990.
- [Ben87] John K. Bennett. The design and implementation of distributed smalltalk. In *Proc of OOPSLA '87*, pages 318–330, Orlando, Florida, October 1987.
- [Ber90] Brian N. Bershad. High performance cross-address space communication. Technical Report 90-06-02, University of Washington, June 1990.
- [BFH<sup>+</sup>90] Bharat Bhargava, Karl Friesen, Abdelsalam Helal, Srinivasan Jagannathan, and John Riedl. Design and implementation of the Raid V2 distributed database system. Technical Report CSD-TR-962, Purdue University, March 1990.
- [Bha87] Bharat Bhargava, editor. *Concurrency Control and Reliability in Distributed Systems*. Van Nostrand and Reinhold, 1987.
- [BJ87] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.
- [BMR87] Bharat Bhargava, Tom Mueller, and John Riedl. Experimental analysis of layered Ethernet software. In *Proc of the ACM-IEEE Computer Society 1987 Fall Joint Computer Conference*, pages 559–568, Dallas, Texas, October 1987.
- [BMR90] Bharat Bhargava, Enrique Mafla, and John Riedl. Communication in the Raid distributed database system. *Computer Networks and ISDN Systems*, December 1990.
- [BR89a] Bharat Bhargava and John Riedl. A model for adaptable systems for transaction processing. *IEEE Transactions on Knowledge and Data Engineering*, 1(4), December 1989.
- [BR89b] Bharat Bhargava and John Riedl. The Raid distributed database system. *IEEE Transactions on Software Engineering*, 15(6), June 1989.
- [BZ91] Bharat Bhargava and Yongguang Zhang. Adaptable communication facility for distributed transaction processing system. Technical Report CSD-TR-91-006, Purdue University, January 1991.
- [CD85] David R. Cheriton and Stephen E. Deering. Host groups: A multicast extension for datagram internetworks. In *Proc of the 9th Data Communication Symposium*, pages 172–179, New York, September 1985.
- [Cha84] Jo-Mei Chang. Simplifying distributed database systems design by using a broadcast network. In *Proc of the ACM SIGMOD Conference*, June 1984.

- [Che84] David R. Cheriton. An experiment using registers for fast message-based inter-process communication. *Operating System Review*, 18(4):12-20, October 1984.
- [Che86] David R. Cheriton. VMTP: A transport protocol for the next generation of communication systems. In *Proc of the SIGCOMM'86 Symposium*, pages 406-415, August 1986.
- [CLZ87] David D. Clark, Mark L. Lambert, and Lixia Zhang. NETBLT: A high throughput transport protocol. In *Proc of the SIGCOMM Conference*, August 1987.
- [DJA88] Partha Dasgupta, Richard J. LeBlanc Jr., and William F. Appelbe. The CLOUDS distributed operating system: Functional description, implementation details and related work. In *Proc of the 8th Intl Conf on Distributed Computing Systems*, San Jose, CA, June 1988.
- [Duc89] Dan Duchamp. Analysis of transaction management performance. In *Proc of the 12th ACM Symposium on Operating Systems Principles*, pages 177-190, Litchfield Park, AZ, December 1989.
- [HPAO89] Norman C. Hutchinson, Larry L. Peterson, Mark B. Abbott, and Sean O'Malley. RPC in the *x*-kernel: Evaluating new design techniques. In *Proc of the 12th ACM Symposium on Operating Systems Principles*, pages 177-190, Litchfield Park, AZ, December 1989.
- [JC89] Gary M. Johnston and Roy H. Campbell. An object-oriented implementation of distributed virtual memory. In *Proc of the USENIX Workshop on Experiences with Distributed and Multiprocessor Systems*, pages 39-57, Fort Lauderdale, FL, October 1989.
- [KTHB89] M. Frans Kaashoek, Andrew S. Tanenbaum, Susan Flynn Hummel, and Henri E. Bal. An efficient reliable broadcast protocol. *Operating Systems Review*, 23(4):5-19, October 1989.
- [LCJS87] Barbara Liskov, Dorothy Curtis, Paul Johnson, and Robert Scheifler. Implementation of Argus. In *Proc of the 11th ACM Symposium on Operating Systems Principles*, November 1987.
- [LMKQ89] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison Wesley, 1989.
- [LZCZ86] Edward D. Lazowska, John Zahorjan, David R. Cheriton, and Willy Zwaenepoel. File access performance of diskless workstations. *ACM Transactions on Computer Systems*, 4(3):238-268, August 1986.
- [Maf90] Enrique Maffa. *Efficient Communication Support for Distributed Transaction Processing*. PhD thesis, Purdue University, December 1990.
- [MB90] Enrique Maffa and Bharat Bhargava. Implementation and performance of a communication facility for distributed transaction processing. Technical Report CSD-TR-1046, Purdue University, November 1990.

- [MSM89] P. M. Melliar-Smith and L. E. Moser. Fault-tolerant distributed systems based on broadcast communication. In *Proc of the 9th International Conference on Distributed Computing Systems*, pages 129–134, Newport Beach, CA, June 1989.
- [PA89] Marc F. Pucci and J. L. Alberti. Experiences with efficient interprocess communication in DUNE. In *Proc of the USENIX Workshop on Experiences with Distributed and Multiprocessor Systems*, pages 349–360, Fort Lauderdale, FL, October 1989.
- [RR81] Richard F. Rashid and George G. Robertson. Accent: A communication oriented network operating system kernel. In *Proc of the 8th Symposium on Operating Systems Principles*, pages 64–75, Pacific Grove, California, December 1981.
- [SB90] Michael D. Schroeder and Michael Burrows. Performance of firefly RPC. *ACM Transactions on Computer Systems*, 8(1):1–17, February 1990.
- [Spe86] Alfred Z. Spector. Communication support in operating systems for distributed transactions. In *Networking in Open Systems*, pages 313–324. Springer Verlag, August 1986.
- [STP<sup>+</sup>87] Alfred Z. Spector, Dean Thompson, Randy F. Pausch, Jeffrey L. Eppinger, Dan Duchamp, Richard Draves, Dean S. Daniels, and Joshua J. Block. CAMELOT: A distributed transaction facility for MACH and the Internet — An interim report. Technical Report CMU-CS-87-129, Department of Computer Sciences, Carnegie Mellon University, June 1987.
- [Svo86] Liba Svobodova. Communication support for distributed processing: Design and implementation issues. In *Networking in Open Systems*, pages 176–192. Springer Verlag, August 1986.
- [VM90] Paulo Veríssimo and José Alves Marques. Reliable broadcast for fault-tolerance on local computer networks. In *Proc of the 9th Symposium on Reliable Distributed Systems*, pages 54–63, Huntsville, Alabama, October 1990.
- [vRvST88] Robbert van Renesse, Hans van Staveren, and Andrew S. Tanenbaum. Performance of the world's fastest distributed operating system. *Operating System Reviews*, 22(4):25–34, October 1988.
- [YTR<sup>+</sup>87] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proc of the 11th Symposium on Operating Systems Principles*, pages 63–76, Austin, TX, November 1987.
- [Zwa85] Willy Zwaenepoel. Protocols for large data transfers over local networks. In *Proc of the 9th Data Communications Symposium*, pages 22–32, Whistler Mountain, British Columbia, Canada, September 1985.





# Experience with Threads and RPC in Mach

Dan Duchamp<sup>1</sup>

Computer Science Department

Columbia University

New York, NY 10027

djd@cs.columbia.edu

## ABSTRACT

Undertaking the design of a multi-threaded event-driven program requires the programmer to grapple not only with how to control concurrency, but also with mechanisms and policies specified by underlying software, including the scheduler, the RPC stub compiler, and of course the thread facility itself. This paper relates the experience acquired with Mach's implementation of these functions while writing a particular multi-threaded program. Advice is given to future designers of both multi-threaded applications and underlying systems software.

## 1. Introduction

This paper relates experience gained while writing a multi-threaded event-driven program using a "threads package" that provides simple thread-manipulation and locking facilities. The threads package is not integrated with a language, but is implemented as a set macros and procedures written in the implementation language (C), calling primitives of the operating system (Mach version 2.0).

This paper examines three versions of the program:

1. Unthreaded. The first version was unthreaded partly because the threads package was not ready when implementation began and partly to reduce the number of ideas that had to be simultaneously tackled.
2. The first threading attempt aiming for high concurrency. This attempt was unsuccessful.
3. A second, less ambitious — and successful — threading attempt.

The implementation environment is discussed first, followed by a description of the design of the unthreaded program (Section 3) and an explanation of how the design was adapted for concurrency (Section 4). Section 5 discusses the failures of this second version and how the failures were repaired in the third version, and mentions some surprises encountered. Section 6 is a summary of the lessons learned from this experience.

---

<sup>1</sup>This work was supported by IBM and the Defense Advanced Research Projects Agency, ARPA Order No. 4976, monitored by the Air Force Avionics Laboratory under Contract F33615-84-K-1520.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any of the sponsoring agencies or of the United States Government.

## 2. Implementation Environment

The particular program in question is TranMan, the transaction manager of the Camelot distributed transaction processing facility [10]. A transaction manager is the component of a transaction processing facility whose prime function is to execute the protocols that ensure the atomic behavior of a multi-process transaction despite the occurrence of any number of failures. TranMan is a separate process acting as a server in the client/server sense.

To use Camelot, someone who possesses a database that he wishes to make publicly available writes a *data server* process that controls the database and allows access to client *application* processes. Any computer on which a data server runs must also run a single instance of each of several processes that comprise the implementation of Camelot, one of which is TranMan. An application initiates a transaction by getting a transaction identifier from the transaction manager and then performs data manipulation operations by making synchronous inter-process procedure calls to any number of data servers, local or remote. The transaction identifier must be explicitly listed as one of the arguments. While processing a request, a data server may in turn call other data servers. Eventually, the application orders the transaction manager to either commit or abort. To commit (abort), the transaction manager must first execute a commit (abort) protocol and then place a commitment (abort) indicator into stable storage.

In Camelot, transactions can be nested according to the so-called “Moss model” [18, 16]. A nested transaction can abort without preventing its parent from later committing. However, the effects of a committed nested transaction are visible only to its parent; furthermore, they are not permanent until the top-level (i.e., non-nested) transaction commits. Rules govern how and when parent and child transactions can access data locked by one another [18]. All the descendants of a top-level transaction — including itself — are said to form a *family*.

Because of its interaction with other processes and other sites, the transaction manager interacts a good deal with the operating system. After describing the key features of Mach, this section describes the RPC stub compiler and the interface of the “C-Threads” [6] threads package used to provide concurrent operation.

### 2.1. Operating System

Camelot runs on the Mach operating system [1]. Mach is a communication-oriented extension of Berkeley UNIX 4.3. The added functions include: the ability to send typed messages addressed to *ports* owned by local processes, read-write sharing of arbitrary regions of memory (among processes that are related as ancestor and descendant), provision of multiple threads of control within a single address space, and an *external pager interface*, to allow a user-mode process to provide virtual memory management for arbitrary regions of memory. Additionally, the Mach implementation has been reorganized (with the addition of locking) to permit it to run on multiprocessors.

Threads are scheduled separately and preemptively. Separate scheduling means that a thread that blocks for some reason (e.g., a page fault) does not prevent the other threads of the same process from executing. The operating system provides no means for synchronization except by performing a blocking system call to receive a message. A message is a sequence of data elements, each element typed according to a simple model that allows integers, strings, real numbers, and ports. The message passing primitives are send, receive (blocking or not), and synchronous send-receive. These primitives are typical of modern message-passing operating systems. [5, 19]

## 2.2. RPC Stub Compiler

Inter-process procedure call is implemented on top of the message system, typically using the MIG (“Mach Interface Generator”) language and stub generator [14].

On the client side, MIG performs the usual marshaling and unmarshaling of arguments. By default, RPC reply messages are sent to a port that is separate for every thread and every interface.

The control flow of a service process is influenced by the function and structure of the server-side stub, which encourages writing the main loop of a server as a receive-compute-send cycle. MIG produces a routine named “foo\_server,” for use by a server implementing interface “foo,” which conveniently hides the details of the message format. The routine takes an input message and a pointer to a buffer for an output message as its two arguments. After a request message is received, foo\_server is called. It unpacks the input message, dispatches to the proper service routine, and packs the response into the output buffer. The buffer is overwritten every time this function (which is typically the main loop of a server) executes. The main loop then sends the reply message.

## 2.3. Threads Package

The thread primitives of Mach 2.0 are purposely quite simple. Mach does not bind an execution stack to a thread; a thread is only a scheduling unit and for all practical purposes is nothing more than a program counter. So programming directly on top of Mach’s thread facility would be difficult. As a consequence, the C-Threads package was written to provide a friendlier thread facility for C programmers.<sup>2</sup>

C-Threads defines data types for threads, locks, and condition variables, and provides routines for manipulating all of these types; use of machine-specific instructions and kernel traps is hidden within the implementation of C-Threads. The important thread manipulation calls are summarized in Figure 2-1, while Figure 2-2 lists the operations on locks and condition variables.<sup>3</sup> The semantics of condition variables are patterned after those of Mesa [15]: signaling a condition does not necessarily result in the immediate execution of a waiting thread, and a broadcast operation is provided.

Condition variable operations are implemented using the message system: condition\_wait is a receive, while condition\_signal and condition\_broadcast are sends. Locks are exclusive, and a thread waits for a lock by spinning. The method for indicating whether a lock is held is unsophisticated: a global integer is either 0 or 1; therefore, a thread can deadlock with itself by requesting a lock that it already holds. This mix of features is found in other thread (or “lightweight process”) packages [2, 3].

## 2.4. Performance Background

Many of the design decisions explained in the coming text were motivated by performance concerns. In Camelot, minimal single-site transactions execute in 13ms on an IBM RT PC, which is a 4-MIP machine. Therefore, expenses down to the millisecond level are considered important. This section gives the measured performance of some basic primitives. Table 2-1 presents Mach performance, while Table 2-2 shows the performance of the basic C-Threads primitives.

---

<sup>2</sup>All of Camelot is written in C.

<sup>3</sup>The material in these figures is taken from from [6].

At the time TranMan was being written Mach was still being tuned and performance was changing almost daily. Nevertheless, the relative performance of several primitives was steady, and there was conscious effort to avoid:

- C-Thread fork/join.
- Condition variable operations.
- Allocation and deallocation of ports.

---

```

/* Definition of generic name for any type of C pointer. */
typedef {...} any_t;

/* Definition of thread data type. */
typedef {...} pthread_t;

/* Start running a thread which will execute procedure "func" with
 * the single argument "arg," which is a pointer of some kind.
 * Return the name of the thread. */
pthread_t pthread_fork(func, arg)
    any_t (*func)();
    any_t arg;

/* Have thread "t" execute synchronously. Do not resume the calling thread until
 * pthread_join returns. */
any_t pthread_join(t)
    pthread_t t;

/* Have thread "t" execute asynchronously (in parallel) with the calling thread.
 * Typically, a detached thread never joins. */
void pthread_detach(t)
    pthread_t t;

/* Terminate the calling thread, returning pointer "results." The termination
 * of procedure "func" — given as the argument to pthread_fork — is translated
 * into pthread_exit(). */
void pthread_exit(results);
    any_t results;

```

---

**Figure 2-1: Thread Manipulation Primitives of the C-Threads Package**

TEST	LATENCY
lock/unlock	6.43
fork/join	1680
condition variable	1680
fork/yield	1580
pthread_self	9.90

**Table 2-2: Times for C-Threads Primitives**

All times are in microseconds. The lock/unlock test gets then drops a single lock. The fork/join test forks a thread and then waits for it to terminate; the thread executes a null procedure. The condition variable benchmark consists of two threads alternatively waiting on and signaling a single condition variable. The fork/yield number indicates how long is needed to fork a thread then yield to the scheduler; each newly forked thread does the same. The pthread\_self test determines how long it takes a thread to learn its id.

---



---

```

/* Definition of lock data type. */
typedef {...} mutex_t;

/* Definition of condition variable data type. */
typedef {...} condition_t;

/* Wait for a lock. */
void mutex_lock(m)
    mutex_t m;

/* Attempt to get a lock: return of 1 means the lock is obtained, 0 means it is not. */
int mutex_try_lock(m)
    mutex_t m;

/* Drop a lock. */
void mutex_unlock(m)
    mutex_t m;

/* Unlock lock "m" then wait for condition "c." */
void condition_wait(c, m)
    condition_t c;
    mutex_t m;

/* If one or more threads is waiting for condition "c," wake up one of them at
 * any convenient time. */
void condition_signal(c)
    condition_t c;

/* If one or more threads is waiting for condition "c," wake up all of them at
 * any convenient time. */
void condition_broadcast(c)
    condition_t c;

```

---

**Figure 2-2: Lock and Condition Variable Primitives of the C-Threads Package**

BENCHMARK DESCRIPTION	TIME
Procedure call, single 32-byte argument	12.0 us
Kernel call, getpid()	149 us
Port allocate & deallocate	1.3 ms
Local IPC, MIG-to-MIG, 8-byte in-line data	1.5 ms
RPC, MIG-to-MIG, 8-byte in-line	19.1 ms
UDP datagram delivery	9.8ms

**Table 2-1: Results of Simple Benchmarks on IBM RT PC**

Measured on 4-MIP IBM RTs connected by a 4Mb/sec IBM token ring.

---

### 3. Unthreaded Transaction Manager

The transaction manager allocates transaction identifiers, keeps track of which servers and remote sites participate in which transactions, and drives execution of the commit and abort protocols. Its most important exported calls are: begin-transaction, commit-transaction, abort-transaction, and join-transaction. Each is synchronous. The join-transaction operation is invoked by a server to inform its local transaction manager that it is participating in the transaction.

### 3.1. Structure

The transaction manager is event-driven, and is implemented in four conceptual layers:

1. The main loop and one decoding and error-checking module for each interface.
2. Stateless modules that contain the logic for various protocols.
3. One module encapsulating each major data structure.
4. Miscellaneous utility routines, including datagram send/receive code.

There are three sorts of events: messages that are part of inter-process procedure calls, network datagrams<sup>4</sup> sent by other transaction managers, and timeout events taken from an internal priority queue. Timeout events indicate what to do if an expected datagram from another site does not arrive in time.

The service paradigm is an augmentation of the basic receive-compute-send cycle. TranMan assigns transaction management client (applications and servers) a private service port; this expense is justified because applications and servers are created only occasionally. The main loop then calls the synchronous receive system call, waiting for a message arriving on any of the service ports or on the port dedicated to communication between transaction managers. The length of time the receive call should block is the amount of time remaining until the next timeout event; if there is no event, then the wait is forever. When a message is received, it is dispatched to the module for the corresponding interface. The interface module then calls modules in lower layers to service the message. After servicing, sometimes the main loop is instructed to send a reply message, sometimes not.

### 3.2. Inter-TranMan Communication

Transaction managers at different sites execute distributed commit/abort protocols by sending datagram messages among themselves, not by using RPC. The original motivation for this decision was speed, since when TranMan was first written a pair of datagrams was faster than an RPC. Later, as indicated by the numbers in Table 2-1, RPC was optimized and datagram communication was not and so RPC is presently a little bit faster.

However, there is an enduring reason for eschewing RPC: the standard RPC paradigm does not support parallel one-to-many communication. During commit, the transaction manager needs to make any number of calls (in parallel) for each of any number of transactions. An attempt to use RPC would fall into one of these categories:

- One call per thread: RPC maps well to this use, but then many threads would be forked and joined during each distributed commit/abort.
- Many calls per thread, each with a separate reply port. In this case, an “RPC” would be split into a send portion and a receive portion, thus breaking the abstraction. Furthermore, before performing the receive call to capture replies, it would be necessary to call into Mach<sup>5</sup> to specify the set of ports on which the replies are expected. This call, which manipulates port rights within the kernel, is expensive and would add several milliseconds to each receive.
- Many calls per thread, with a single reply port per transaction: with this method the use of `port_set_add` is no longer necessary, but the RPC abstraction remains unused and the demultiplexing chore is the same as with datagrams.

---

<sup>4</sup>Use of datagrams rather than RPC for TranMan-TranMan communication is motivated in the next section.

<sup>5</sup>via what is now named the `port_set_add` call

After these considerations, and bearing in mind that at the moment of the original design datagram communication was faster than RPC, the decision was made to have a single datagram port for the entire TranMan and to multiplex/de-multiplex all messages through this port.

The right abstraction would be “parallel RPC.” With parallel RPC TranMan-TranMan communication could have been made conceptually simpler; the only tricky point would have been arranging per-site rather than per-call timeout intervals. There is of course existing work on parallel RPC and related ideas [17, 20, 7].

### 3.3. Algorithms

The transaction manager is essentially a finite-state protocol machine; i.e., the algorithms it executes are communication protocols for coordinating the actions of several (possibly distributed) processes. These include the well-known two-phase commitment protocol [11], a fault-tolerant commitment protocol [8], and an abort protocol that prevents the commitment of orphans [9]. Every protocol has a “table-based” implementation in a separate module in layer 2; that is, there is (conceptually) a separate procedure specifying the action to take for each combination of protocol, transaction-state, and input event.

### 3.4. Data Structures

The important data structures inhabiting layer 3 of the transaction manager are:

- Hash table of family descriptors.
- Forest of transaction descriptors rooted by descriptors of top-level transactions, plus hashed access to every descriptor.
- Priority queue of pending timeout events.
- List of active application processes.
- List of active data server processes.

Family and transaction descriptors are separate data structures because a nested transaction can abort (and disappear) without its parent having to abort. Roughly speaking, family and transaction descriptors store the information needed for commit and abort, respectively.

Inserts and lookups keyed by transaction identifier must be fast, so hashed access is provided to both family and transaction descriptors. The speed of deletion is not so important. Other requirements are that it should be quick to go from a family descriptor to the corresponding transaction descriptor (for abort and deletion), and to go between child and parent transaction (for abort). These requirements motivated the creation of trees of transaction descriptors. Each transaction descriptor is linked to a hash bucket and to its parent, first child, and siblings in the family tree.

## 4. Initial Adaptation to Multi-threading

There are two motivations for multi-threading a transaction manager. The first is to improve throughput: even on a uniprocessor, it is highly desirable to be able to continue to operate while some threads are performing long, synchronous operations such as a disk write. Second, multi-threading provides an elegant method for exploiting parallelism while running on multiprocessors.

#### 4.1. Sources of Parallelism

Common patterns of transaction usage suggest three increasingly fine possible grains of parallelism within transaction management:

1. Inter-family parallelism: separate invocations of the transaction manager may run simultaneously only if the transactions are in different families. Since commitment is the longest operation, inter-family parallelism is useful mostly for committing several families at the same time.
2. Intra-family parallelism: in addition to inter-family parallelism, different transactions within the same family may run simultaneously within TranMan. Intra-family parallelism is beneficial if nested transactions are common.
3. Complete intra-transaction parallelism: all TranMan calls may proceed in parallel. Examples include processing the join-transaction messages from two servers for the same transaction.

Although Camelot's current workload consists mostly of small non-nested transactions, in anticipation of the day when nested transactions might be common the initial threading attempt included a locking scheme intended to permit intra-family parallelism. Operations representing intra-transaction parallelism are rare and too fine-grained to justify the scheduling of a Mach thread, so no attempt was made to exploit intra-transaction parallelism.

#### 4.2. Problems and Solutions

The need to meet certain performance goals placed constraints on how threads were added to TranMan. First, there were some obvious goals: set locks to provide for intra-family parallelism, minimize the superfluous holding of locks, and avoid the use of the generally expensive C-Threads thread manipulation primitives. These constraints affect the design of linked data structures and lock placement. Second, to simplify datagram de-multiplexing, the decision was made to have only one datagram port for the entire transaction manager. This constraint affects the allocation of work to threads.

The basic approach to handling multiple threads was the following:

- Use locks to protect critical regions that manipulate primary data structures. Avoid sharing message and log buffers by having a set for each thread.
- Create a pool of threads when the process starts and increase the number as needed. Never destroy a thread.
- When executing a distributed protocol, do not "tie" any thread to any particular function or transaction. Instead, have every thread wait for any type of input, process the input, and resume waiting.

These points are amplified in the next three subsections, with a fourth subsection giving attention to the added problem of implementing timeouts.

##### 4.2.1. Lock Placement

The action of getting or dropping a lock is a side effect. A key design principle was to minimize the number of procedures performing side effects: those that get a lock but do not also drop it, and vice versa. Obvious exceptions are routines for creation, lookup, and deletion of descriptors.

Most data structures were easily adapted to concurrency. Buffers for log records and messages are allocated on the per-thread stack, and so are not data structures that need to be shared among threads. The priority queue and the lists of applications and data servers are accessed only very briefly within a single procedure each, so in each case having a lock for the whole structure was



acceptable.

Read/write locks were seen as a critical need to prevent unnecessary lock contention. They are provided by a second locking package — called “rw-lock” — built on top of C-Threads<sup>6</sup>. Besides eliminating the obvious waste of having an exclusive lock where only a read lock is needed, rw-lock uses condition variable signaling instead of spinning to implement waiting. Figure 4-1 shows the implementation of rw-lock.

Implementing inter-family parallelism requires having separately locked family descriptors. For intra-family parallelism, transaction descriptors must also be separately locked. In both hash tables a read/write lock was added to each bucket of the hash anchor table. The lock must be obtained in write mode in order to insert or delete a descriptor, while lookup (by far the most common operation) requires only a read lock. Each descriptor is protected from simultaneous updates by an exclusive lock embedded within it. Like many other programs, the TranMan relies on the natural distribution of load by hashing to ensure that there is little lock contention. Tests of the frequency of contention verify that it is indeed the case the contention is rare, even at high load.

The method for deadlock avoidance is classic: there is a defined order of locks, and when a thread is to hold several locks simultaneously, it must obtain the locks in order [13]. The order is: hash bucket, descriptor, others. To avoid multi-process deadlocks, all locks are dropped before TranMan calls any other process.

#### 4.2.2. Thread Creation

Because it is hard to predict the number of threads needed, and in order to avoid thread-creation overhead on incoming requests, TranMan allocates a pool of threads when it starts. Thereafter, a thread is created only if all other threads are busy. Threads are never deallocated. To do so would be a poor decision. The only cost of having too many is the stack space used, while a policy of thread deallocation mixed with a bursty workload could lead to “thread thrashing.” The main loop — including the code that creates and uses threads — is given in Figure 4-2.

#### 4.2.3. Thread Allocation

If generating the reply to a particular service request requires first sending and receiving other messages or datagrams, then either the servicing thread must use condition variables to pause and be resumed later when the response is ready (the “pause/resume approach”), or else the buffer containing the forming reply must be saved so that another thread can send it later (the “delayed send approach”). The single-thread version of the transaction manager did not have this choice. It had to use the delayed send approach.

The pause/resume method is more elegant, but has several disadvantages:

1. It does not map as naturally to a “event-driven, table-based” implementation of a protocol.
2. The mechanism for pausing and resuming threads (a condition variable implemented with Mach messages) makes pause/resume somewhat slower.
3. Many more threads may be simultaneously active.

A subtle advantage is that the thread ID provides an easy unique identifier for an operation. This comes in handy for executions of certain protocols that — unlike commit and abort — can be invoked any number of times by the same transaction. For these protocols the transaction iden-

---

<sup>6</sup>The author is Josh Bloch.

tifier alone cannot be used to identify the operation.

It is not clear which method is better. The transaction manager employs a hybrid approach in which pause/resume is used for handling inputs that do not require executing a distributed protocol, and the delayed send method is used for distributed protocols.

#### 4.2.4. Timeouts

Timeouts represented a problem. The unthreaded transaction manager performed a synchronous receive call, blocking until the moment of the earliest expected timeout. When the call returned, either a new input message had been received or it was time to perform some timeout recovery action. Extending this approach to a multi-threaded design would have all threads receive, waiting until the same moment. If the timeout did occur, all threads would wakeup despite there being work for only one. In the more common case where a message arrived and caused some thread to cancel the timeout, then all waiting threads would either have to adjust their wait intervals or awake with no work to do.

TranMan dedicates one thread (the *timeout thread*) to waiting for timeout events. All the remaining threads (*general-purpose threads*) are available to service any message event. The timeout thread waits by performing a timed receive operation on a single dedicated port, the *timeout port*. The general-purpose threads wait forever on all other ports. These ports are: one port for receiving datagrams, one for messages from the kernel, one for every application or data server, one for every Camelot process, and one extra called the *event service port*. Having all threads wait on all ports is conceptually simple, and avoids the trouble and performance degradation of having a mechanism within TranMan that acts as a work multiplexor. Instead, work requests (messages) go directly to threads.

Whenever a timeout happens, the timeout thread wakes up, removes the timeout event description from the priority queue, and sends a special message describing the event to the event service port. One of the general-purpose threads receives this message<sup>7</sup> and processes the event. This scheme consumes few extra resources: the timeout and event service ports, and the timeout thread. The real issue is the cost of enqueueing or dequeueing a timeout event.

Enqueueing an event is now more complicated than simply adding an item to a priority queue. If the new event becomes the first in the queue, then a message is sent to the timeout port telling the timeout thread to adjust its wait-time. In a lightly loaded system the event queue is typically quite short (often length 0 before the enqueue), and a message is sent often (e.g., when the event queue increases from 0 to 1 event).

Dequeueing is also somewhat complicated. If dequeueing removes the first event in the queue, no special action is taken. The timeout-thread will timeout early, see that there is no work to be done, and reset its timeout interval correctly after examining the queue. Since nearly every enqueue leads to a dequeue and not a timeout, this scenario will usually take place.

---

<sup>7</sup>When several threads are waiting for a message on one or more ports, the thread whose receive call succeeds is selected by Mach in round-robin fashion.

---

```

typedef struct rw_lock {
    struct mutex latch; /* Latch to synchronize lock ops */
    struct condition read_free; /* Indicates lock is free for readers */
    int nreaders; /* Number of read locks held */
    struct condition write_free; /* Indicates lock is free for writers */
    int write; /* Flag to indicate write lock held */
    int nwrite_waiters; /* Number of threads waiting to write */
} *rw_lock_t;

/* rw_read_lock -- Obtain a shared lock on the given lock variable. If the
 * lock is unavailable, the calling thread will block until the lock becomes
 * available. */
#define rw_read_lock(l)
do {
    mutex_lock(&(l)->latch);
    while ((l)->write || (l)->nwrite_waiters != 0)
        condition_wait(&(l)->read_free, &(l)->latch);
    (l)->nreaders++;
    mutex_unlock(&(l)->latch);
} while(0)

/* rw_write_lock -- Obtain an exclusive lock on the given lock variable. If
 * the lock is unavailable, the calling thread will block until the lock
 * becomes available. */
#define rw_write_lock(l)
do {
    mutex_lock(&(l)->latch);
    (l)->nwrite_waiters++;
    while ((l)->nreaders != 0 || (l)->write)
        condition_wait(&(l)->write_free, &(l)->latch);
    (l)->nwrite_waiters--;
    (l)->write = TRUE;
    mutex_unlock(&(l)->latch);
} while(0)

/* rw_unlock -- Releases the lock previously obtained via rw_lock (or
 * rw_try_lock). It is crucial that this procedure be called only if the
 * lock is actually held. */
#define rw_unlock(l)
do {
    mutex_lock(&(l)->latch);
    if ((l)->write)
        (l)->write = FALSE;
    else
        (l)->nreaders--;
    if ((l)->nreaders == 0)
        if ((l)->nwrite_waiters != 0)
            condition_signal(&(l)->write_free);
        else
            condition_broadcast(&(l)->read_free);
    mutex_unlock(&(l)->latch);
} while(0)

```

---

**Figure 4-1: Read/Write Locks Built Using C-Threads**

A new lock type (`rw_lock_t`) is defined which permits multiple simultaneous readers. Waiting (for either read or write access) is done with `condition_wait()`. When the lock is dropped, either a single writer is awakened with `condition_signal()`, or all readers are awakened with `condition_broadcast()`.

---

---

```

static int Initialize()
{
    int i;

    LOCK(&threadCountLock);
    for (i = 0; i < InitialThreadNumber; i++)
        pthread_detach(pthread_fork(MainLoop));
    threadsFree = InitialThreadNumber;
    threadsExisting = InitialThreadNumber;
    UNLOCK(&threadCountLock);
}

static int MainLoop()
{
    msg_t inM, outM;

    while (1) do {
        /* wait forever to receive message */
        MSG_RECEIVE(&inM, FOREVER);

        /* mark thread as busy; or create a new one if none left */
        LOCK(&threadCountLock);
        assert(threadsFree >= 1);
        if (threadsFree > 1) {
            threadsFree--;
            UNLOCK(&threadCountLock);
        } else {
            threadsExisting++;
            UNLOCK(&threadCountLock);
            pthread_detach(pthread_fork(MainLoop));
        }

        /* dispatch and send reply */
        if (ta_server(&inM, &outM) /* application interface */
            || ts_server(&inM, &outM) /* server interface */
            || tc_server(&inM, &outM) /* ComMan interface */
            || td_server(&inM, &outM) /* DiskMan interface */
            || tr_server(&inM, &outM) /* RecMan interface */
            || tt_server(&inM, &outM) /* datagrams */
            || event_dispatch(&inM, &outM)) /* timeout event */
        {
            if (outM.retCode != DO_NOT_REPLY)
                MSG_SEND(&outM);
        } else {
            ERROR(("unknown message (id %d) from port %d",
                inM.msg_id, inM.msg_remote_port));
        }

        /* thread now free */
        LOCK(&threadCountLock);
        threadsFree++;
        UNLOCK(&threadCountLock);
    }
}

```

---

**Figure 4-2: Multi-Thread Main Loop**

The statement “`pthread_detach(pthread_fork(MainLoop))`” in routine “Initialize” creates a new thread that executes the standard receive-compute-send cycle, “MainLoop.” Variable “threadsExisting” records the number of threads; similarly, “threadsFree” records the number of unallocated threads. Whenever MainLoop begins to process an input, it decrements threadsFree. If the last free thread is about to be allocated, another thread is created.

---



## 5. The Final Version

The problems with the first threaded version of TranMan were few, but severe. These troubles and their solutions are reviewed in the following text.

### 5.1. Locking

By far the most serious deficiency of the initial threaded version was the deadlocks resulting from overly ambitious and ill-designed lock placement. During deletion of transaction descriptors, a recursive procedure traversed the links of transaction trees from the top down; to perform deletion, this procedure had also to lock the hash buckets of the descriptors it visited. While traversing deep trees, occasionally two transaction descriptors would be hashed to the same bucket. Since all locks remained held during the recursion, trying to lock the bucket for the second time would cause self-deadlock. There were a number of solutions to this dilemma:

- Redo the lock package to permit certain locks to be re-acquired by a thread. This solution can be easily programmed on top of C-Threads, but at a measurable cost to the very common operation of getting a lock. Self-deadlock could be ruled out more efficiently within the C-Threads implementation, but this would have reopened a significant C-Threads design decision.
- Redesign either the deletion procedure or the lock acquisition method. For example, one option is to record in a bucket which thread has it locked, and check this record before trying to get a lock. This approach would slow the process of acquiring a bucket lock (a very common operation) to guard against a very uncommon bug.
- Redo lock placement to permit only inter-family concurrency. Little is lost by implementing only inter-family parallelism and requiring intra-family operations to serialize — common operations for nested transactions are quick and hold locks for only a short time.

The third approach was taken, resulting in new data structures as shown in Figure 5-1. Now, each family houses a private hash table of the descriptors of transactions within the family. A single lock protects the family descriptor and all the transaction descriptors for that family. The single large hash table containing all transaction descriptors is eliminated. Within each private transaction hash table there still must be pointers from child to parent, sibling to sibling, and parent to first child. The possibility of deadlock is eliminated because access to any descriptor within a family is serialized. Only uncommon combinations of operations theoretically suffer contention. These combinations are mostly cases where many parallel nested transactions are simultaneously doing the same thing: all joining, all committing, and so on.

Three lessons are evident here:

1. *Trying to lock “2D” data structures is hard and perhaps not worthwhile.* In the TranMan, one “dimension” is formed by the vertical tree links, the other by the horizontal hash bucket links.
2. *Recursion while holding locks is dangerous.* The recursive deletion procedure that was a simple and powerful tool in the single-threaded transaction manager generated unbounded deadlock opportunities when used in conjunction with 2D locking. More generally, recursion is a good way of having a thread create the nested monitor problem [12] with itself.
3. *Re-acquiring an already-held lock is not always a bug.* Some locking facilities (e.g., Sun’s lightweight process library [21]) allow this, some do not. Those that do typically do so on a per-lock basis. The ideal facility might allow this capability on a per-use basis.

	Lock for data fields and private hash table of transaction records
Data fields	Data fields
Link to first transaction record within transaction hash table	Link to PRIVATE hash table of transactions within this family
Hash bucket links	Hash bucket links

**Figure 5-1:** Comparison of Family Descriptor Before and After Multi-threading

The record on the left is how the family descriptor was defined in the single-threaded version of the transaction manager. The multi-thread version, on the right, contains a new field (the lock) and a pointer to a per-family hash table of transaction descriptors rather than a pointer into a hash table of all transaction descriptors.

## 5.2. Scheduling

Having multiple active threads naturally gives rise to a scheduling problem. Locks and condition variables are in fact nothing but simple scheduling mechanisms usable by the programmer rather than the kernel. Pernicious interactions between locking and preemptive scheduling are well known; e.g., the “convoy phenomenon” [4] and priority-caused starvation [15, 3].

TranMan exhibited a new scheduling anomaly resulting in the creation of many (e.g., 25 to 35) threads even at light loads (i.e., only 1 thread needed). The cause was Mach’s “hand-off scheduling” mechanism. Hand-off scheduling causes a sending thread to yield the processor immediately (i.e., within the same timesharing quantum) to the destination thread, provided that the destination thread is blocked receiving. The intent of hand-off scheduling is to accomplish transfer of the processor with as few system entries as possible. Doing hand-off during message send optimizes the Mach scheduler for inter-process procedure call: the receiver runs as soon as its message is available.

With multi-threaded programs, the following scenario can occur during a single time quantum:

- A. TranMan sends reply #1 to its client.
- B. Before the send call returns, the client is immediately scheduled.
- C. Client sends request #2 to TranMan, using Mach’s send-and-receive system call.  
The client blocks awaiting the reply.
- D. TranMan is scheduled, receives and processes the request, and sends reply #2.
- E. Step B repeats, etc.

The code in Figure 4-2 will create a new thread within TranMan every time a new request arrives in step D. The ping-pong scheduling of TranMan and a persistent client (and the consequent uncontrolled creation of threads within TranMan) will continue until hysteresis functions become active within Mach, causing unfinished send calls — such as those in step A — to return.

Hand-off scheduling is an effective technique for speeding up IPC. However, for a busy process that is often “IPC bound,” hand-off scheduling can cause the number of active threads to explode much higher than the expected degree of multiprocessing. Since in the current C-Threads implementation a thread stack occupies 100K, gross over-allocation of threads will make a process’ working set quite a bit larger than necessary.

This problem was never truly fixed: the “solution” was to place a (high) cap on the number of threads allowed to be active. This solution caused no immediate problems because the transaction manager is not a throughput bottleneck, but it obviously does not scale well to multiprocessors.

Another scheduling-related annoyance was the inability to create low-priority “background” garbage collection threads to scavenge things such as deallocated family descriptors. The Mach thread manipulation system calls permit a thread to adjust its priority, but C-Threads does not make this capability visible.

These observations coupled with the others mentioned above suggest that perhaps thread facilities and the operating systems supporting them should have some means to coordinate scheduling policies.

### 5.3. Remote Procedure Call

The combination of the Mach message passing primitives, C-Threads, and the MIG RPC stub compiler combined to generate two sorts of inconveniences. First, MIG does not produce a server stub procedure that does nothing but pack an output message. The lack of such a capability makes the delayed send approach messy to implement. The server must allocate a buffer for each delayed response and duplicate the code for packing the response. This code contains the message type identifier and other details that should be hidden from the programmer. Once, message type identifiers changed without this section of the program being updated. The result was that TranMan started sending confusing response messages. This particular bug spurred the change to the pause/resume approach of thread allocation.

Second, while it is elegant to have all threads wait for work by calling the blocking receive system call, timeout processing requires the ability to quickly unblock a thread. Had there existed a fast mechanism for interrupting a blocked thread, a wider array of design choices would have existed for handling timeouts. Apollo’s Concurrent Programming Support threads package has an interrupt mechanism called `$task_signal` [2] that can be used for this purpose.

## 6. Summary of Lessons Learned

Below is an enumeration of some opinions regarding programming with multiple threads that were generated while building TranMan. The first two are programming hints that, while unoriginal, are easily forgotten. The next three statements are one user’s pleas to designers of future thread facilities. Items 6 and 7 are simple statements about RPC support, while item 8 is a issues for research.

1. A hash table is an effective source of parallelism that is simple to lock correctly. Be firmly convinced of the need for greater concurrency before you design a more elaborate data structure.
2. A recursive, lock-acquiring procedure can be dangerous.
3. Re-acquiring an already-held lock is not always a bug.
4. Permit a thread to affect its priority.

5. Allow a thread to be awakened while blocked in (at least some) system calls.
6. RPC stub compilers should support reply messages by supplying a response-packing stub procedure. Simply because a remote procedure call appears synchronous to the client should not constrain the server to a synchronous implementation.
7. Parallel RPC (such as described in [17, 20]) seems very important for properly structuring “truly distributed” (as opposed to client/server) computations.
8. Design unified scheduling mechanisms to permit operating system scheduling policies to work in cooperation — not competition — with scheduling policies implicitly specified within multi-threaded programs by the use of locks, condition variables, and priority changes.

## Code Availability

Mach and Camelot can be obtained from the Mt. Xinu corporation.

## Acknowledgements

Dean Thompson and Elliot Jaffe produced the third version of the transaction manager and helped discover some of the observations made here. Eric Cooper answered my questions about details of C-Threads. Ed Lazowska, Tom Anderson, Brian Bershad, and Gail Kaiser provided criticism that improved the presentation.



## References

- [1] M. Accetta, et. al.  
Mach: A New Kernel Foundation for UNIX Development.  
In *Proc. of Summer Usenix*, pages 93-112. July, 1986.
- [2] *Concurrent Programming Support Reference*  
Apollo Computers, Inc., Chelmsford, MA, 1987.
- [3] A. D. Birrell.  
*An Introduction to Programming with Threads.*  
Technical Report 35, Digital Systems Research Center, January, 1989.
- [4] M. Blasgen, J. Gray, M. Mitoma, T. Price.  
The Convoy Phenomenon.  
*Operating Systems Review* 13(2):20-25, April, 1979.
- [5] D. R. Cheriton.  
The V Distributed System.  
*Comm. ACM* 31(3):314-333, March, 1988.
- [6] E. C. Cooper, R. P. Draves.  
*C Threads.*  
Technical Report CMU-CS-88-154, Carnegie Mellon University, June, 1988.
- [7] E. C. Cooper.  
Programming Language Support for Multicast Communication in Distributed Systems.  
In *Proc. Tenth Intl. Conf. on Distributed Computing Systems*, pages 450-457. IEEE, May, 1990.
- [8] D. Duchamp.  
*A Non-blocking Commitment Protocol.*  
Technical Report CUCS-457-89, Columbia Univ. Computer Science Dept., August, 1989.  
Submitted for publication.
- [9] D. Duchamp.  
*An Abort Mechanism for Nested Distributed Transactions.*  
Technical Report CUCS-459-89, Columbia Univ. Computer Science Dept., November, 1989.  
Submitted for publication.
- [10] D. Duchamp, et. al.  
*Design Rationale of the Camelot Distributed Transaction Facility.*  
Technical Report CUCS-008-90, Columbia Univ. Computer Science Dept., March, 1990.  
Submitted for publication.
- [11] J. N. Gray.  
Notes on Database Operating Systems.  
In R. Bayer, R. M. Graham, G. Seegmuller (editors), *Lecture Notes in Computer Science.*  
Volume 60: *Operating Systems - An Advanced Course*, pages 393-481. Springer-Verlag, 1978.
- [12] B. Haddon.  
Nested Monitor Calls.  
*Operating Systems Review* 11(4):18-23, October, 1977.

- [13] J. W. Havender.  
Avoiding Deadlock in Multitasking Systems.  
*IBM Systems Journ.* 7(2):74-84, April, 1968.
- [14] M. B. Jones, R. F. Rashid, M. R. Thompson.  
Matchmaker: An Interface Specification Language for Distributed Processing.  
In *Proc. Twelfth Ann. Symp. on Principles of Programming Languages*, pages 225-235.  
ACM, January, 1985.
- [15] B. Lampson and D. Redell.  
Experience with Processes and Monitors in Mesa.  
*Comm. ACM* 23(2):105-117, February, 1980.
- [16] B. Liskov.  
Distributed Programming in Argus.  
*Comm. ACM* 31(3):300-313, March, 1988.
- [17] Bruce Martin.  
*Parallel Remote Procedure Call Language Reference and User's Guide*  
Univ. of California, San Diego, San Diego, 1986.
- [18] J. E. B. Moss.  
*Nested Transactions: An Approach to Reliable Distributed Computing*.  
MIT Press, 1985.
- [19] S. J. Mullender, A. S. Tanenbaum.  
The Design of a Capability-Based Distributed Operating System.  
*The Computer Journal* 29(4):289-300, 1986.
- [20] M. Satyanarayanan and E. H. Siegel.  
Parallel Communication in a Large Distributed Environment.  
*IEEE Trans. Computers* 39(3):328-348, March, 1990.
- [21] *Sun OS 4.0 Reference Manual*  
Sun Microsystems, Inc., Palo Alto, CA, 1987.

# Kernel-Kernel Communication in a Shared-Memory Multiprocessor<sup>†</sup>

Eliseu M. Chaves, Jr.\*

Thomas J. LeBlanc

Brian D. Marsh

Michael L. Scott

Computer Science Department

University of Rochester

Rochester, New York 14627

(716) 275-9491

{chaves,leblanc,marsh,scott}@cs.rochester.edu

## Abstract

In the standard kernel organization on a shared-memory multiprocessor all processors share the code and data of the operating system; explicit synchronization is used to control access to kernel data structures. Distributed-memory multicomputers use an alternative approach, in which each instance of the kernel performs local operations directly and uses remote invocation to perform remote operations. Either approach to inter-kernel communication can be used in a NonUniform Memory Access (NUMA) multiprocessor, although the performance tradeoffs may not be apparent in advance.

In this paper we compare the use of remote access and remote invocation in the kernel of a NUMA multiprocessor operating system. We discuss the issues and architectural features that must be considered when choosing an inter-kernel communication scheme, and describe a series of experiments on the BBN Butterfly designed to empirically evaluate the tradeoffs between remote invocation and remote memory access. We conclude that the Butterfly architecture is biased towards the use of remote invocation for most kernel operations, but that a small set of frequently executed operations can benefit from the use of remote access.

## 1. Introduction

An important consideration in the design of any multiprocessor operating system kernel is the host architecture, which will often dictate how kernel functionality is distributed among processors, the form of inter-kernel communication, the layout of kernel data structures, and the need for synchronization. For example, in uniform memory access (UMA) multiprocessors, it is easy for all processors to share the code and data of the operating system. Explicit synchronization can be used to control access to kernel data structures. Both distributed-memory multicomputers (e.g., hypercubes and mesh-connected machines) and distributed systems use an alternative organization, wherein the kernel data is distributed among the processors, each of which executes

---

<sup>†</sup> This research was supported by NSF grant no. CCR-9005633, NSF Institutional Infrastructure grant no. CDA-8822724, a DARPA/NASA Graduate Research Assistantship in Parallel Processing, the Federal University of Rio de Janeiro, and the Brazilian National Research Council.

\* Visiting Scientist on leave from the Universidade Federal do Rio de Janeiro, Brazil.

a copy of the kernel. Each kernel performs operations on local resources directly and uses remote invocation to request operations on remote resources. Nonpreemption of the kernel (other than by interrupt handlers) provides implicit synchronization among the kernel threads sharing a processor.

Although very different, these two organizations each have their advantages. A shared-memory kernel is similar in structure to a uniprocessor kernel, with the exception that access to kernel data structures requires explicit synchronization. As a result, it is straightforward to port a uniprocessor implementation to a shared-memory multiprocessor.<sup>1</sup> Having each processor execute its own operations directly on shared memory is also very efficient. In addition, this kernel organization simplifies load balancing and global resource management, since all information is globally accessible to all kernels.

Message-passing (i.e., remote invocation) kernels, on the other hand, are naturally suited to architectures that don't support shared memory. Each copy of the kernel is able to manage its own data structures, so the source of errors is localized. The problem of synchronization is simplified, since all contention for data structures is local, and can be managed using nonpreemption. This kernel organization scales easily, since each additional processor has little impact on other kernels, other than the support necessary to send invocations to one more kernel.

NonUniform Memory Access (NUMA) multiprocessors, such as the BBN Butterfly [2], IBM 8CE [9], and IBM RP3 [15] have properties in common with both shared-memory multiprocessors and distributed-memory multicomputers. Since NUMA multiprocessors support both remote memory access and remote invocation, kernel data can be accessed using either mechanism. The performance tradeoffs between the use of remote invocation and remote access in the kernel of a NUMA machine are not well understood, however, and depend both on the specific architecture and on the overall design of the operating system.

In this paper we explore the tradeoffs between remote access and remote invocation in the kernel, and the related issues of locality, synchronization, and contention. Our observations about the tradeoffs are made concrete through a series of experiments comparing the direct and indirect costs associated with each design decision. We conclude with a summary of the relationship between architectural characteristics and kernel organization for NUMA multiprocessors.

## 2. Kernel-Kernel Communication Options

We consider a machine organization consisting of a collection of *nodes*, each of which contains memory and one or more processors. Each processor can access all the memory on the machine, but it can access the memory of the local node more quickly than the memory of a remote node. When a processor at node *i* begins executing an operation that must access data on node *j*, interaction among nodes is required. There are three principal classes of implementation alternatives:

### remote memory access

The operation executes on node *i*, reading and writing node *j*'s memory as necessary. The memory at node *j* may be mapped by node *i* statically, or it may be mapped on demand.

---

<sup>1</sup> Several versions of Unix have been ported to multiprocessors simply by protecting operating system data structures with semaphores [3]. An alternative approach is to use a master/slave organization wherein all kernel calls are executed on a single node; other nodes contain only a trap handler and a remote invocation mechanism. BBN's nX version of Unix uses this approach, as do the Unix portions of Mach [1], from which nX was derived.



#### remote invocation

The processor at node  $i$  sends a message to a processor at node  $j$ , asking it to perform the operation on its behalf.

#### bulk data transfer

The kernel moves the data required by the operation from node  $j$  to node  $i$ , where it is inspected or modified, and possibly copied back. The kernel programmer may request this data movement explicitly, or it may be implemented transparently by a lower-level system using page faults.

In evaluating the tradeoffs between these three options, we distinguish between *node locality* and *address locality*. Address locality captures the traditional notion of spatial locality in sequential programs. We say that a program (e.g. the kernel) displays a high degree of address locality if most of the memory locations accessed over some moderate span of time lie within a small set of dense address ranges. Node locality, by contrast, captures the notion of physical locality in a NUMA multiprocessor. We say that the kernel displays a high degree of node locality if most operations can be performed primarily using local memory references on some node.

Whatever the mechanism(s) used to communicate between instances of the kernel, performance clearly depends on the ability to maximize node locality. Any operating system that spends a large fraction of its time on operations that require interaction between nodes is unlikely to perform well. It seems reasonable to expect, and our experience confirms, that a substantial amount of node locality can in fact be obtained. This implies that most memory accesses will be local even when using remote memory accesses for kernel-kernel communication, and that the total amount of time spent waiting for replies from other processors when using remote invocation will be small compared to the time spent on other operations.

At the same time, experience with uniprocessor operating systems suggests that it is very hard to build a kernel with a high degree of address locality. There are several reasons for this difficulty. Kernels operate on behalf of a potentially large number of user processes, whose actions are generally unrelated to each other. To the extent that they are related, the most pronounced effect is likely not to be continuity of working set across context switches, but rather fragmentation of the working set of any particular process, as it incorporates common data structures. Typical kernel construction techniques rely heavily on pointer-based linked data structures, the pieces of which are often dynamically allocated.

In terms of the communication options listed above, the lack of address locality in the kernel suggests that data accessed by any particular kernel operation are unlikely to be in physical proximity, casting doubt on the utility of bulk data transfer for the implementation of kernel-kernel communication. Of course, we have not deliberately attempted to organize data structures to maximize address locality in any of the systems we have built, nor are we aware of any attempts to do so in other projects. It is therefore possible that the lack of address locality is simply an artifact of avoidable design decisions. Large parts of the PLATINUM kernel [6] are implemented on top of a "coherent memory" system that replicates and migrates data in response to page faults. Experiments with PLATINUM may eventually lead to a better understanding of the utility of bulk data transfer in the kernel.

In the remainder of this section, and in the case study that follows, we focus on the choice between remote memory access and remote invocation. We consider direct, measurable costs of individual remote operations, indirect costs imposed on local operations, the effects of competition among remote operations for processor and memory cycles, and the extent to which different communication mechanisms complement or clash with the structural division of labor among processes in the kernel. Ultimately, we argue in favor of a mixture of both mechanisms, since no one mechanism will be the best choice for all operations.

## 2.1. Direct Costs of Remote Operations

A reasonable first cut at deciding between remote memory access or remote invocation for a particular operation can be made on the basis of the latency incurred under the two different implementations. For example, consider an operation  $O$  invoked from node  $i$  that needs to perform  $n$  memory accesses to a data structure on another node  $j$ . We can perform those memory accesses remotely from node  $i$ , or we can perform a remote invocation to node  $j$ , where they will be performed locally. For the sake of simplicity, suppose that  $O$  must perform a fixed number of local memory accesses (e.g. to stack variables) and a fixed number of register-register operations regardless of whether it is executed on node  $i$  or on node  $j$ . If the remote/local memory access time ratio is  $R$  and the overhead of a remote invocation is  $C$  times the local memory access time, then it will be cheaper to implement  $O$  via remote memory access when  $(R-1)n < C$ .

The fixed overhead of remote invocation, independent of operation complexity, suggests that operations requiring a large amount of time should be implemented via remote invocation (all other things being equal).<sup>2</sup> Back of the envelope calculations should suffice in many cases to evaluate the performance tradeoff. Many operations are simple enough to make a rough guess of memory access counts possible, and few are critical enough to require a truly definitive answer. For critical operations, however, experimentation is necessary.

## 2.2. Indirect Costs for Local Operations

An important factor that we ignored in the above comparison based on latency is that operations will often be organized differently when performed via remote invocation. They may require context available on the invoking node to be packaged into parameters. They may be reorganized in order to segregate accesses to data on the invoking processor into code that can be executed before or after the remote invocation. Most important, perhaps, the use of remote invocation for *all* accesses to a particular data structure may allow that data structure to be implemented without explicit synchronization, depending instead on the implicit synchronization available via lack of context-switching as in a uniprocessor kernel. Although preemption is still possible from interrupt handlers, the cost of disabling interrupts is typically much lower than the cost of explicit synchronization.

Avoiding explicit synchronization can improve the speed not only of the remote operations but also of the (presumably more frequent) local operations that access the same data structure. The impact of explicit synchronization on local operations is easy to underestimate. We will see operations in our case study in which lock acquisition and release account for 49% of the total execution time (in the absence of contention). This overhead could probably be reduced by a coarser granularity of locking, but only with considerable effort: fine-grain locking requires less thought and allows greater concurrency.

On a machine in which individual nodes are multiprocessors (with parallel execution of one local copy of the kernel), explicit synchronization may be required for certain data structures even if remote invocation is always used for operations on those data structures requested by other nodes. On the other hand, clever use of fetch-and- $\Phi$  operations to create concurrent no-wait data structures [10, 13] may allow explicit synchronization to be omitted even for data structures whose operations are implemented via remote memory access.

---

<sup>2</sup> We did not include the cost of parameter passing in our simple analysis. Nearly all our kernel operations take only one parameter, and the reply value is used to signal completion of the operation, so our assumption of a fixed cost for remote invocation is realistic.

If remote memory accesses are used for many data structures, large portions of the kernel data space on other processors will need to be mapped into each instance of the kernel. Since virtual address space is limited, this mapping may make it difficult to scale the kernel design to very large machines, particularly if kernel operations must also be able to access the full range of virtual addresses in the currently-running user process. Mapping remote kernel data structures on demand is likely to cost more than sending a request for remote invocation. Mechanisms to cache information about kernel data structures may be limited in their effectiveness by the lack of address locality. Systems that map kernel-kernel data into a separate kernel address space [16] may waste large amounts of time switching back and forth between the kernel-kernel space and the various user-kernel spaces.

### 2.3. Competition for Processor and Memory Cycles

Operations that access a central resource must serialize at some level. Operations implemented via remote invocation serialize on the processor that executes those operations. Operations implemented via remote memory accesses serialize at the memory. Because an operation does more than access memory, there is more opportunity with remote memory access for overlapped computation. Operations implemented via remote memory access may still serialize if they compete for a common coarse-grain lock, but operations implemented via remote invocation will serialize even if they have no data in common whatsoever.

If competition for a shared resource is high enough to have a noticeable impact on overall system throughput it will clearly be desirable to reorganize the kernel to eliminate the bottleneck. The amount of competition that can occur before inducing a bottleneck may be slightly larger with remote memory access, because of the ability to overlap computation. Even in the absence of bottlenecks, we expect that operations on a shared data structure will occasionally conflict in time. The coarser the granularity of the resulting serialization, the higher the expected variance in completion time will be. The desire for predictability in kernel operations suggests that operations requiring a large amount of time should be implemented via remote memory access, in order to serialize at the memory instead of the processor. This suggestion conflicts with the desire to minimize operation latency, as described above; it may not be possible simultaneously to minimize latency and variance.

### 2.4. Compatibility With the Conceptual Model of Kernel Organization

There are two broad classes of kernel organization, which we refer to as the horizontal and vertical approaches. These alternatives correspond roughly to the message-based and procedure-based approaches, respectively, identified by Lauer and Needham in their 1978 paper [11]. In a vertical kernel there is no fundamental distinction between a process in user space and a process in the kernel. Each user program is represented by a process that enters the kernel via traps, performs kernel operations, and returns to user space. Kernel resources are represented by data structures shared between processes. In a horizontal kernel each major kernel resource is represented by a separate kernel process, and a typical kernel operation requires communication (via queues or message-passing) among the set of kernel processes that represent the resources needed by the operation.

Both approaches to kernel organization can be aesthetically appealing, depending on one's point of view. The vertical organization presents a uniform model for user- and kernel-level processes, and closely mimics the hardware organization of an UMA multiprocessor. The horizontal organization, on the other hand, leads to a compartmentalization of the kernel in which all synchronization is subsumed by message passing. The horizontal organization closely mimics the hardware organization of a distributed-memory multicomputer. Because it minimizes context switching, the vertical organization is likely to perform better on a machine with uniform memory [5]. The horizontal organization may be easier to debug [8]. Most Unix kernels are vertical. Demos [4] and Minix [17] are horizontal.



Remote invocation seems to be more in keeping with the horizontal approach to kernel design. Remote memory access seems appropriate to the vertical approach. If porting an operating system from some other environment, the pre-existence of a vertical or horizontal bias in the implementation may suggest the use of the corresponding mechanism for kernel-kernel communication, though mixed approaches are possible [12]. If a vertical kernel is used on a uniprocessor, the lack of context switching in the kernel may obviate the need for explicit synchronization in many cases. Extending the vertical approach to include remote memory access may then incur substantial new costs for locks. On a machine with multiprocessor nodes, however, such locking may already be necessary.

### 3. Case Study: Psyche on the BBN Butterfly

Our experimentation with alternative communication mechanisms took place in the kernel of the Psyche operating system [16] running on a BBN Butterfly Plus multiprocessor [2]. The Psyche implementation is written in C++, and uses shared memory as the primary kernel communication mechanism. The Psyche kernel was modified to provide performance figures for remote invocation as well, with and without fine-grain locking. Our results are based on experiments using these modified versions of the kernel.

The basic abstraction provided by Psyche is the *realm*, a passive object containing code and data. A *process* is a thread of control representing concurrent activity within an application. Processes are created, destroyed, and scheduled by user-level code, without requiring kernel intervention. User-level processes interact with one another by invoking realm operations. Processes are executed by *virtual processors*, or *activations*. The kernel time-slices the processor among the activations located at a node. Activations execute in address spaces known as *protection domains*, and obtain access to realms by means of an *open* operation that maps a realm into the caller's domain.

The implementation of the Psyche abstractions favors node locality. The kernel object representing an application-level abstraction is allocated and initialized on a single node, either on the node where the request originated or another specified node. Other kernel data structures associated with a node's local resources are also local to that node. It is quite common, therefore, for a kernel operation not to need access to data on another node. In those cases where kernel-kernel communication is required, local accesses still tend to dominate.

Among those kernel operations requiring access to data on more than one node, it was common in the original Psyche implementation for remote memory accesses to occur at several different times in the course of the operation. In an attempt to optimize our *protected procedure call* mechanism (a form of RPC) we found that many, though not all, of these accesses could be grouped together by re-structuring the code, thereby permitting them to be implemented by a single remote invocation.

#### 3.1. Fundamental Costs

The Butterfly Plus is a NUMA machine with a remote:local memory access time ratio of approximately 12:1. The average measured execution time [7] of an instruction to read a 32-bit remote memory location using register indirect addressing is 6.88  $\mu$ s; the corresponding instruction to read local memory takes 0.518  $\mu$ s. The time to write memory is slightly lower: 4.27  $\mu$ s and 0.398  $\mu$ s for remote and local memory, respectively.<sup>3</sup> Microcoded support for block copy operations can be used to move large amounts data between nodes in about a fifth of the time

---

<sup>3</sup> The original Butterfly architecture had a remote-to-local access time ratio of approximately 5:1. The speed of local memory was significantly improved in the Butterfly Plus, with only a modest improvement in the speed of remote accesses.



required for a word-by-word copy (345  $\mu$ s instead of 1.76 ms for 1K bytes). None of the experiments reported below moved enough data to need this operation.

Our remote invocation mechanism relies on remote memory access and on the ability of one processor to cause an interrupt on another. A processor that requires a remote operation writes an operation code and any necessary parameters into a preallocated local buffer. It then writes a pointer to that buffer into a reserved location on the remote node, and issues a remote interrupt. The requesting processor then spins on a "operation received" flag in the local buffer. When the remote processor receives the interrupt, it checks its reserved location to obtain a pointer to the buffer. It sets the "operation received" flag, at which point the requesting processor begins to spin on an "operation completed" flag. If another request from a different node overwrites the original request, the second request will be serviced instead. After a fixed period of unsuccessful waiting for the "operation received" flag, the first processor will time-out and resend its request. In case a processor's request is completed just before a resend, receiving processors ignore request buffers whose "operation received" flag is already set.

The remote invocation mechanism is *optimistic*, in that it minimizes latency in the absence of contention and admits starvation in the presence of contention. Its average latency, excluding parameter copying and operation costs, is 56  $\mu$ s. An earlier, non-optimistic, implementation relied on microcoded atomic queues, but these required approximately 60  $\mu$ s for the enqueue and dequeue operations alone.

### 3.2. Explicit Synchronization

Psyche uses spin locks to synchronize access to kernel data structures. In order to achieve a high degree of concurrency within the kernel, access to each component data structure requires possession of a lock. This approach admits simultaneous operations on different parts of the same kernel data structure, but also introduces a large number of synchronization points in the kernel. Opening (mapping) a realm, for example, can require up to nine lock acquisitions. Creating a realm can require 38 lock acquisitions. A cheap implementation of locks is critical.

We use a test-and-test&set lock [14] to minimize latency in the absence of contention. If the lock is in local memory, we use the native MC68020 TAS instruction. Otherwise, we use a more expensive atomic instruction implemented in microcode on the Butterfly (TAS is not supported on remote locations). The slight cost of checking to see whether the lock is local (involving a few bit operations on its virtual address) is more than balanced by the use of a faster atomic primitive in the common, local case.

A lock can be acquired and released manually, by calling inline subroutines, or automatically, using features of C++. The automatic approach passes the lock as an initialization parameter to a dummy variable in the block of code to be protected. The constructor for the dummy variable acquires the lock; the destructor (called by the compiler automatically at end of scope) releases it. Constructor-based critical sections are slightly slower, but make it harder to forget to release a lock. Manual locking is used for critical sections that span function boundaries or that do not properly nest. Acquiring and releasing a local lock manually requires a minimum of 5  $\mu$ s, and may require as much as 10  $\mu$ s, depending on instruction alignment, the ability of the compiler to exploit common subexpressions, and the number of registers available for temporary variables. Acquiring and releasing a remote lock manually requires 38 to 45  $\mu$ s. The additional time required to acquire and release a lock through constructors is about 1 to 3  $\mu$ s. Synchronization using remote locks is expensive because the Butterfly's microcoded atomic operations are significantly more costly than native processor instructions. Extensive use of no-wait data structures [10] might reduce the need for fine-grain locks, but would probably not be faster, given the cost of atomic operations.

### 3.3. Impact on the Cost of Kernel Operations

To assess the impact of alternative kernel-kernel communication mechanisms on the performance of typical kernel operations, we measured the time to perform several such operations via local memory access, remote memory access, and remote invocation, with and without explicit synchronization. The results appear in Table 1. The first three lines give times for low-latency operations. The first of these inserts and then removes an element in a doubly-linked list-based queue; the second and third search for elements in a list. The last three lines give times for high-latency operations: creating a realm, opening (mapping) a realm, and adding a new activation to a protection domain. All times are accurate to about  $\pm 3$  in the third significant digit. Times for the low-latency operations are averaged over 10,000 trials. They are stable in any particular kernel load image, but fluctuate with changes in instruction alignment. They are also sensitive to the context in which they appear, due to variations in the success of compiler optimizations. We have read through the assembly language output of the compiler for our timing tests, to make sure the optimizer isn't removing anything important. Times for the high-latency operations are averaged over 1 to 10 trials. They are limited by the resolution of the 62.5  $\mu$ s clock.

Times in columns 1 and 2 are with all data on the local node. Times in columns 3 through 6 are with target data on a remote node, but with temporary variables still in the local stack. Columns 1 and 3 give times for the unmodified version of the Psyche kernel. Column 2 indicates what operations would cost if synchronization were achieved through lack of context switching, with no direct access to remote data structures. Column 4 indicates what operations on remote data structures would cost if subsumed in some other operation with coarse-grain locking. Column 6 indicates what remote operations would cost if always executed via remote invocation, so that the lack of context-switching would obviate the need for locks. Column 5 indicates the cost of performing operations via remote invocation in a hybrid kernel that continues to rely on locks.

In actuality, of course, the use of remote invocation for all remote operations eliminates the need for true mutual exclusion locks, but retains the problem of synchronization between normal activity and the remote invocation interrupt handler. The times in columns 2 and 6 may therefore underestimate real costs. (The times in column 2 do apply, as shown, to subsumption in larger operations with coarser locking.) A more realistic implementation of remote invocation without explicit locking would employ a bit indicating whether normal execution was currently in the kernel. If the kernel were already active, the remote invocation handler would queue its request

Operation		Local Access		Remote Access		Remote Inv.	
		locking		locking		locking	
		on	off	on	off	on	off
enqueue+dequeue	( $\mu$ s)	42.4	21.6	247	154	197	174
find last in list of 5	( $\mu$ s)	25.0	16.1	131	87.6	115	96.7
find last in list of 10	( $\mu$ s)	40.6	30.5	211	169	125	105
create realm	(ms)	6.20	5.69	14.8	13.1	6.87	6.37
open realm	(ms)	0.96	0.86	3.05	2.62	1.15	1.09
create activation	(ms)	1.43	1.35	3.30	3.04	1.53	1.43

Table 1: Overhead of Kernel Operations

for execution immediately prior to the next return to user space. If the kernel were not active, the interrupt handler could execute its operation immediately, at interrupt level, or it could use a mechanism such as the VAX AST to force a context switch out of user space and into the kernel upon return from the interrupt handler. Execution directly from the interrupt handler is clearly faster, but may or may not be appropriate for high-latency operations. We have used it in all our tests, and our figures indicate the performance that results when the kernel is not otherwise active. With the exception of diagnostic serial lines, devices in the Butterfly are attached to a single "king" node; processors other than the king are in no danger of losing device interrupts due to extended computation at high priority.

### Explicit Synchronization

As seen in Table 1, the cost of synchronization dominates in simple operations on queues, introducing in some cases nearly 100% overhead for local operations and 60% overhead for remote operations. Though less overwhelming, synchronization impacts more complex operations as well, due to the use of fine-grain locks. Realm creation requires acquiring and releasing approximately 38 constructor-based locks, adding over 500  $\mu$ s, or 9%, to the cost in the local case and 1.7 ms, or 13%, to the cost in the remote case. The overhead of fine-grain locking combined with automatically-acquired locks is clearly significant. More to the point, this overhead is imposed on local access to data structures in order to *permit* remote access to those structures. We could reduce the cost of synchronization by locking data structures at a coarser grain. This change would reduce the number of locks required by a typical operation, but would simultaneously reduce the potential level of concurrency.

### Remote References

We can assess the impact of remote memory references by comparing the cost of local and remote operations in Table 1. Without locking, the marginal cost of remote references accounts for 86% of the cost of a remote enqueue/dequeue operation pair; remote references exclusive of synchronization account for 54% of the cost even when locking is used (154  $\mu$ s to perform the operation remotely excluding synchronization costs minus 21.6  $\mu$ s to perform the operation locally, over 247  $\mu$ s total time). When searching for the 10th element in a list, remote references exclusive of synchronization account for 2/3 of the cost of the operation. Even for complex operations such as realm creation, which performs much of its work out of the stack, remote references account for half of the total cost.

The overhead associated with explicit synchronization and remote references is a function of the complexity of the operation, while the overhead associated with remote invocation is fixed. In addition, if using remote invocation exclusively we can rely on implicit synchronization (non-preemption in the kernel), thereby reducing the cost of operations significantly. In table 1, the times in the last three rows of column 6 are not only much faster than the corresponding times in column 3, they are comparable to the times in column 1; the ability to avoid lock acquisition and release hides the cost of remote invocation and parameter passing. The enqueue/dequeue operation and the search in a list of 5 both take less time via remote invocation, without synchronization, than they take via remote memory access with synchronization. If we could avoid the need for synchronization, however, (e.g. by coarse-grain locking), remote access would be cheaper. Since a remote memory access costs more than 6  $\mu$ s more than a local access, and a remote lock/unlock pair costs about 40  $\mu$ s, the 60  $\mu$ s overhead of a remote invocation with a single parameter can be justified to avoid four remote references and a lock/unlock pair. If synchronization were free, remote invocation could still be justified to avoid eleven remote references.



## 4. Conclusions

Architectural features strongly influence operating system design. The choice between remote invocation and remote access as the basic communication mechanism between kernels on a shared-memory multiprocessor is highly dependent on the cost of the remote invocation mechanism, the cost of atomic operations used for synchronization, and the ratio of remote-to-local memory access time. Since the overhead associated with remote access scales with the operation, while the overhead associated with remote invocation is fixed, we would expect remote access to outperform remote invocation only on relatively simple operations. The operating system designer must determine exactly which operations, if any, would benefit from the use of remote access, and whether the impact on the overall design of the operating system would be justified.

On the Butterfly Plus, remote invocation is relatively fast, explicit synchronization is costly, and remote references significantly more expensive than local references. As a result, few operations can be executed more efficiently with remote access than with remote invocation. In fact, remote invocation dominates even if the kernel exhibits node locality. Although originally introduced to minimize remote references, node locality can also ensure that only one remote invocation is required per kernel operation. If no attempt had been made to maximize node locality, no complex operation could have been performed with one remote invocation; many invocations would be needed just to collect the data necessary to perform an operation. Under those circumstances remote access would be competitive; however, the resulting organization would not be a reasonable one for the Butterfly architecture.

Given the obvious advantages of remote invocation on the Butterfly Plus, why did we consider using shared memory in the original Psyche kernel? First, we were attracted to the shared-memory kernel model on aesthetic grounds and believed that remote accesses could be minimized and overall performance made acceptable with an appropriate degree of node locality. We did not realize the extent to which the cost of remote access would be dominated by synchronization, nor did we explicitly recognize that node locality would also improve the performance of remote invocation. We expected that the remote accesses required by a typical operation, even if few in number, would often be separated by significant amounts of local computation, and would therefore require several separate remote invocations. Second, our original experience with remote invocation suggested that it took over 150  $\mu$ s to perform a remote operation, which increased the appeal of remote access. Unfortunately, this experience was based on an implementation that used Butterfly atomic operations extensively. Our current implementation does not use those operations at all. Third, our estimates of the cost of synchronization were based heavily on the efficiency of the MC68020 test&set instruction, and did not sufficiently consider such additional factors as the need to differentiate between local and remote locks, the overhead of constructors and destructors, and the frequency of synchronization.

Despite our conclusions regarding the advantages of remote invocation, remote access can play an important role in the kernel. For example, the PLATINUM kernel on the Butterfly [6] uses remote access to manage page tables and can free a page in 10  $\mu$ s. Efficiency in managing page information is particularly important in PLATINUM because the operating system replicates and migrates pages frequently to create the illusion of uniform access memory. Since most operations on page tables and memory management data structures are simple operations, they are particularly well-suited to remote access. Thus, there clearly is a role for remote access in the kernel. The significance of that role will vary from machine to machine, depending on architectural parameters. On the Butterfly, remote invocation should be the dominant mechanism, reserving remote access for frequently executed, specialized operations.



## Acknowledgments

Our thanks to Rob Fowler for his helpful comments on this paper, and to Tim Becker for his invaluable assistance with experiments.

## References

1. M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian and M. Young, "Mach: A New Kernel Foundation for UNIX Development," *Proc. of the Summer 1986 USENIX Technical Conference and Exhibition*, Pittsburgh, PA, June 1986.
2. BBN Advanced Computers Inc., Inside the Butterfly Plus, Oct 1987.
3. M. J. Bach and S. J. Buroff, "Multiprocessor Unix Systems," *AT&T Bell Laboratories Technical Journal* 63, 8 (Oct 1984), pp. 1733-1750.
4. F. Baskett, J. H. Howard and J. T. Montague, "Task Communication in Demos," *Proc. 6th ACM Symp. on Operating System Principles*, West Lafayette, IN, Nov 1977, pp. 23-31.
5. D. Clark, "The Structuring of Systems Using Upcalls," *Proc. 10th ACM Symp. on Operating System Principles*, Orcas Island, WA, Dec 1985, pp. 171-180.
6. A. L. Cox and R. J. Fowler, "The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with PLATINUM," *Proc. 12th ACM Symp. on Operating System Principles*, Litchfield, AZ, Dec 1989, pp. 32-44.
7. A. L. Cox, R. J. Fowler and J. E. Veenstra, "Interprocessor Invocation on a NUMA Multiprocessor," TR 356, Department of Computer Science, University of Rochester, Oct 1990.
8. R. A. Finkel, M. L. Scott, Y. Artsy and H. Chang, "Experience with Charlotte: Simplicity and Function in a Distributed Operating System," *IEEE Transactions on Software Engineering* 15, 6 (June 1989), pp. 676-685.
9. A. Garcia, D. Foster and R. Freitas, "The Advanced Computing Environment Multiprocessor Workstation," IBM Research Report RC-14419, IBM T.J. Watson Research Center, Mar 1989.
10. M. Herlihy, "A Methodology for Implementing Highly Concurrent Data Structures," *Proc. of the Second PPOPP*, Seattle, WA, Mar 1990, pp. 197-206.
11. H. C. Lauer and R. M. Needham, "On the Duality of Operating System Structures," *Operating Systems Review* 13, 2 (Apr 1979), pp. 3-19.
12. T. J. LeBlanc, J. M. Mellor-Crummey, N. M. Gafter, L. A. Crowl and P. C. Dibble, "The Elmwood Multiprocessor Operating System," *Software—Practice & Experience* 19, 11 (Nov 1989), pp. 1029-1056.
13. J. M. Mellor-Crummey, "Concurrent Queues: Practical Fetch-and-Phi Algorithms," TR 229, Department of Computer Science, University of Rochester, Nov 1987.
14. J. M. Mellor-Crummey and M. L. Scott, "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors," *ACM Transactions on Computer Systems*, to appear. Earlier version published as TR 342, Department of Computer Science, University of Rochester, April 1990, and COMP TR90-114, Center for Research on Parallel Computation, Rice University, May 1990.

15. G. R. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton and J. Weiss, "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture," *Proc. 1985 International Conference on Parallel Processing*, St. Charles, IL, Aug 1985, pp. 764-771.
16. M. L. Scott, T. J. LeBlanc, B. D. Marsh, T. G. Becker, C. Dubnicki, E. P. Markatos and N. G. Smithline, "Implementation Issues for the Psyche Multiprocessor Operating System," *Computing Systems* 3, 1 (Winter 1990), pp. 101-137.
17. A. S. Tanenbaum, *Operating Systems: Design and Implementation*, Prentice-Hall, Englewood Cliffs, NJ, 1987.

# Process Scheduling and Synchronization in the *Renaissance* Object-Oriented Multiprocessor Operating System

Vincent F. Russo  
Department of Computer Science  
Purdue University  
West Lafayette, IN 47906  
russo@cs.purdue.edu

## Abstract

This paper relates experience and knowledge gained in the construction of the *Renaissance* object-oriented multiprocessor operating system under development at Purdue University. *Renaissance* is a new multiprocessor object-oriented operating system that grew out of the author's experiences with the *Choices* system from the University of Illinois at Urbana-Champaign. This paper concentrates specifically on how well, and efficiently, object-oriented programming and design techniques support the construction of parallel operating system software. In particular, I focus on process scheduling and synchronization in a multiprocessor system. The specific algorithms used in the *Renaissance* system are discussed along with their implementation and performance. Alternate algorithms, including those used in *Choices* are discussed as well.

## 1 Introduction

*Renaissance* is a new multiprocessor object-oriented operating system under development at Purdue University. It grew out of the author's experience with the *Choices* system from University of Illinois at Urbana-Champaign. *Renaissance* is a reinvestigation of the ideas and algorithms learned in *Choices* and an extension of those ideas into a distributed object environment. *Renaissance* is intended to be a platform upon which to conduct distributed and multiprocessor operating system research. In particular, the *Renaissance* kernel discussed in this paper will form the basis for a complete distributed object system. Extending the *Choices* goal of providing system objects to applications transparently, the complete *Renaissance* system will provide transparent access to remote objects distributed throughout a network of machines.

*Renaissance* is an object-oriented operating system [Rus91] designed and constructed using object-oriented techniques. In object-oriented design and programming, an *object* is the basic entity of abstraction. An object is a set of state or *instance* variables and a set of operations or *methods* that update and access its state. The methods defined by an object provide the only means by which other objects in the system can access the object's instance variables and perform operations upon them.

In order to simplify programming commonalities between objects, each object in an object-oriented system is defined as an *instance* of some *class*. A class defines an interface and specifies the implementation of the methods in that interface for all objects which are instances of the class. A class can be thought of as a *template* or *cookie-cutter* for objects.

When object-oriented programmers describe their systems, they refer to invoking operations on objects as sending *messages* to those objects. These *message sends* cause corresponding *methods* of the object to be invoked. This message-passing paradigm may sound expensive at first glance, but in reality object method invocation can be very light-weight. No actual "messages" are usually exchanged. In compiled object-oriented languages like C++[Str86], each object has what amounts to a pointer to its class. Each class has a table of the location of all its methods. A run-time determination of the proper method to call for a message send can, therefore, be quickly done by indexing into this table. The method is then invoked with a normal procedure call. Since C++ is statically typed, the offset into the table can be determined at compile time. When the compiler knows the exact type of the receiver of the message, the method lookup can even be avoided and, in the best case, the method call expanded in-line. *Renaissance* is implemented in C++ to obtain this efficiency.

The interface and implementation defined by a class need not always be fully specified. Part or all of it can be taken from other classes. Obtaining pieces of a class's interface or implementation from other classes is referred to as *inheritance*. Classes providing inherited parts are usually called *parent* or *super* classes. The new, inheriting class is usually referred to as a *subclass*. Repeated subclassing leads to a *class hierarchy* which successively refines knowledge about the system. Inheritance allows specification of only the differences between a new class and some existing classes to be needed before a class with a desired functionality can be created. The object-oriented principle of *polymorphism* allows class inheritance to be useful. Polymorphism is the ability of a function to take arguments of many different types. In particular, a method can take objects that are instances of many different classes. Class inheritance and polymorphism make object-oriented programming an ideal model for both code and interface reuse. *Renaissance* takes advantage of both extensively.

Within a class hierarchy, some classes serve only as placeholders for implementation or interface descriptions and are never themselves instantiated. Rather, they are subclassed to fill in missing details and these subclasses are instantiated. Such a placeholder class will be termed an *abstract* class. An *abstract* class specifies a general signature (the methods that may be used on instances of the class and its concrete subclasses) and partial implementation of the signature. A *concrete* class refines the implementation of an abstract class. Abstract classes are used in object-oriented system design to specify abstractions. Concrete subclasses are used to specify particular versions, policies or mechanisms that implement the abstraction. For this reason, the name of an abstract class is often used collectively to refer to the type of instances of any of its concrete subclasses. This convention will be used in the rest of this paper.

In an object-oriented system, all system entities are modeled as objects instantiated from representative classes. In a truly object-oriented *operating system*, all machine dependencies, operating system mechanisms and policies, and design decisions are encapsulated within classes. This means that everything from page tables and device registers, to processes and files should be described by classes and encapsulated by representative objects. A full system is realized by a framework that describes how these classes interact.

## 2 Process Management in *Renaissance*

Details of process management for an operating system are very low-level and architecture-specific. They provide an excellent demonstration of how object-oriented programming



can support such low-level details without sacrificing performance even in a multiprocessor environment. The goal of the *Renaissance* process management system is to support an efficient implementation of a model of an application composed of a potentially large number of parallel processes that can share portions (or all) of their address spaces through mechanisms described in [RC89].

## 2.1 Overview

The concept of a *process* is fundamental to all modern operating systems. A process represents a program in execution[PS85]. An *individual control path* through a program in execution is perhaps a more precise definition since a program may have multiple concurrent execution paths. In traditional systems, an individual process follows a control path between a program's various functions and procedures. An object-oriented system is characterized by messages being sent to objects in order to perform computation. These message sends cause object methods to be invoked. Therefore, with respect to object-oriented systems, a process follows a control path between object methods. In both object-oriented and traditional systems, each process has a *current execution point* (the address of the instruction it is currently executing). The current execution point of a process in a traditional system is always within a particular procedure or function. In an object-oriented system the current execution point is always within a particular method. Message sends cause the current execution point to move from method to method of various objects.

A process is characterized by an address space describing the memory it can access and the state of the processor that is executing it. This state is usually referred to as the process's *context*. While a process is active on a processor, its context is reflected in that processor. When the process is not active on a processor, its context must be saved. In order to transfer the processor between various processes, an operating system must implement context switching primitives to allow it to switch the physical processor between the contexts of different processes.

In the *Renaissance* process model, all processes are represented as objects (instances of the **Process** class, or any subclasses). A process's execution is manipulated by sending messages to such an object. In particular, context switching is implemented by sending messages to **Processes**. Examples of objects that would send such messages to a process include those implementing semaphores, monitors, and time-sharing schedulers.

## 2.2 Process Management Classes

*Renaissance* divides the functions of process management and scheduling between three major abstract classes.<sup>1</sup>

1. The **Process** class represents a process and its context.
2. The **Processor** class represents a physical processor.
3. The **ProcessContainer** class represents a repository of **Processes**.

A unique **Process** object represents each process.<sup>2</sup> The **Process** class defines the `giveProcessorTo` message to implement context switching. This message is sent from within

<sup>1</sup>These classes are refinements of classes originally introduced by *Choices*.

<sup>2</sup>The distinction between the term process with a capital P (**Process**) and a without (process) is that the former will be used when referring to an instance of a class with the **Process** signature while the latter will be used to refer to the logical entity that it represents.

synchronization or scheduling objects to the current **Process** in order to effect processor sharing.<sup>3</sup> The **giveProcessorTo** message takes a single argument: the next **Process** to run.

Primitives for scheduling and blocking processes are built using instances of classes in the **ProcessContainer** hierarchy. A **ProcessContainer**, as the name implies, is a repository of **Processes**. Scheduling decisions involve transferring **Processes** between **ProcessContainers** and switching the processor to the contexts of processes that are removed from **ProcessContainers**. As will be discussed in Section 2.6, transferring processes between containers can lead to many pitfalls in a multiprocessor environment.

*Renaissance* avoids a special scheduling process by supporting primitives to directly exchange the processor between processes. Eventually, either involuntarily as the result of an exception or voluntarily, a process invokes **giveProcessorTo** to cause a context switch to another process. For example, consider implementing a semaphore. When a semaphore P operation is performed, if the semaphore is busy, the currently executing process needs to be blocked and another process run[Dij68]. This occurs in *Renaissance* by the current process placing itself in a queue of processes blocked on the semaphore, choosing another process to run from the queue of ready processes, and sending the **giveProcessorTo** message to itself with the new **Process** to run as an argument. Both the queue of processes blocked on the semaphore, and the queue of processes ready to run are represented by **ProcessContainers**. When a V operation is performed on the semaphore, a blocked process can be removed from the semaphore's blocked process queue and added to the queue of ready processes.

These process management classes are presented in detail in the following sections. Special attention is paid to how object-oriented programming and design have benefited its design and construction as well as supported optimizations of the model in a clean and modular manner.

## 2.3 The Process Class

**Process** is an abstract class. Subclasses of **Process** represent different kinds of processes. Each subclass reflects the requirements of the kind of process it represents. For example, **ApplicationProcesses** represent processes executing user programs, and **SystemProcesses** represent operating system management processes. Other subclasses exist to represent processes serving specialized operating system functions. For example, the process running at the time the system is booted (**BootProcess**) or the process executed when there are no other processes to run (**IdleProcess**).

Each **Process** forms a repository for a process's context while the process is not active on any processor. Likewise, a **Process** is the object to which messages are sent to alter a process's context. For example, a **Process** can be sent messages to disable or enable interrupts, or to set scheduling parameters or priorities. For the sake of efficiency, the amount of information kept per-process, and the context switching effort between two processes, is minimized. The context switching overhead is based on the kind of process relinquishing the processor and the kind of process being given the processor.

When a new process is created, the corresponding **Process** is parameterized by an address space, a stack size, an initial execution address, and arguments to the procedure at this address. Each process shares memory with other processes through the memory management mechanisms described in [RC89]. These mechanisms allow arbitrary shared/private regions to be set up between cooperating processes.

<sup>3</sup>The **thisProcess()** function exists to obtain a reference to the current **Process**.

The context of a process takes a different form on every computer architecture. Usually it consists of a set of register contents, a program counter and a stack pointer. In order to increase portability by localizing architecture dependencies in as few places as possible, the state of a process is actually split between two objects, a **Process** and a **ProcessorContext**. **Processes** encapsulate all of the processor architecture *independent* information about a process and is portable across architectures. This information consists mainly of scheduling parameters and a reference to the process's address space information. The processor architecture *dependent* context of a process is kept in an associated **ProcessorContext**. **ProcessorContext** defines messages to be sent during context switching to save (the **checkpoint** message) and restore (the **restore** message) the processor architecture dependent context of a process. Subclasses of **ProcessorContext** represent the saved context of a process for specific architectures. Subclasses of these classes further refine the **ProcessorContext** **checkpoint** and **restore** methods to handle specific kinds of **Processes** on these architectures.

Since many **Process** methods, for example enabling and disabling interrupts (the **enableInterrupts** and **disableInterrupts** methods), are architecture dependent, they are actually implemented in terms of **ProcessorContext** methods.

## 2.4 The Processor Class

A physical processor in *Renaissance* is represented by an instance of the **Processor** class.<sup>4</sup> Multiprocessors are handled by having multiple instances of **Processor**, one per physical processor. An executing process can use the **thisProcessor()** function to obtain a reference to the **Processor** object managing the processor on which it is currently executing. **Processor** is an abstract class subclassed for each physical processor type to which the system is targeted. In addition to process scheduling, it provides other processor specific functions such as finding the handler for a particular trap or interrupt.

## 2.5 The ProcessContainer Class

The **ProcessContainer** class forms the basis of the *Renaissance* process scheduling system. **ProcessContainer** is an abstract class defining a signature for storing and retrieving **Processes**. This signature includes the messages **add** (for inserting **Processes** into the container), **remove** (for removing **Processes** from the container), and **isEmpty** (for testing whether or not the container contains any **Processes**). Concrete subclasses of **ProcessContainer** impose different queuing disciplines on the processes that they contain by implementing the **add** and **remove** methods to (for example) add **Processes** and remove them in FIFO or priority order.

Process scheduling in *Renaissance* follows the basic running-ready-blocked model[Dei84], but is supported within the object-oriented framework *Renaissance* provides. All queues of processes are represented as **ProcessContainers**. In the running-ready-blocked model, all processes that could execute as soon as a processor is free are kept in the *ready-queue*. In *Renaissance*, the job of the ready-queue is potentially split among multiple **ProcessContainers** to avoid contention and to provide process scheduling flexibility.

<sup>4</sup>The corresponding class was actually called **CPU** in *Choices* but the functionality is almost identical.

### 2.5.1 The Per-Processor Container

Multiple ready-queues are supported in *Renaissance* and each processor may be assigned a different (or shared) queue. Having multiple ready-queues allows the partitioning of processes among groups of processors in a multiprocessing system. Each **Processor** references a **ProcessContainer** as its ready queue.

Sending the **getNextReadyProcess** message to a **Processor** returns the next process from that **Processor**'s ready-queue. Scheduling and synchronization objects send this message to select another process to run if they are not relinquishing the processor to a predetermined process. For example, when a process blocks on an I/O request or a semaphore.

If the processor's ready-queue is empty, **getNextReadyProcess** returns the **Processor**'s **idleProcess**, which is a process that is always ready to execute. A **Processor**'s **idleProcess** merely loops with interrupts enabled until another process is ready to run (until the ready-queue is no longer empty). When another process is ready to run, the **idleProcess** relinquishes the processor to that process by sending itself **giveProcessorTo**.

### 2.5.2 The Per-Process Container

Since it is desired to support multiple ready-queues, a mechanism is needed to decide in which queue a newly ready process belongs. This is solved by each **Process** maintaining a reference to the ready-queue to which it will be added when it is ready to execute. A process's ready-queue is initially set to its creating process's value.

Sending a **Process** the **ready** message sets its state to ready and adds the **Process** to its ready-queue as shown in the implementation of the **ready** method below:

```
Process::ready()
{
    assert( ( state == Blocked ) || ( state == Running ) );
    state = Ready;
    myReadyQ->add( this );
}
```

After a **Process** is constructed and initialized, it is sent the **ready** message to allow it to run as soon as a processor becomes available. The **ready** message is also sent to unblock a blocked process after an I/O event completes or when a busy semaphore becomes free.

The complement of the **ready** message in *Renaissance* is the **block** message:

```
Process::block()
{
    assert( state == Running );
    state = Blocking;
    Process * nextProcess = thisProcessor()->getNextReadyProcess();
    giveProcessorTo( nextProcess );
}
```

A process sends itself the **block** message when it has placed itself in a container other than a ready-queue (for example, a semaphore block queue) and desires to relinquish the processor. The **block** method sends the current **Processor** the **getNextReadyProcess** message and resumes that process. It is the responsibility of another process to eventually remove the



blocking process from the container it was placed in at a later time and send it the *ready* message. In *Renaissance*, both the *ready* and *block* messages are available to application programmers to allow them direct control over process scheduling within their applications. For example, an application programmer can implement a user-level semaphore with these primitives. Currently, the *giveProcessorTo* message is not available to application programs. The potential usefulness of this is currently being investigated.

### 2.5.3 Scheduling Ready Processes

The *Renaissance* scheduling model can be adapted for many needs. A traditional symmetric multiprogrammed system has a single ready-queue. This is achieved in *Renaissance* by having all **Processes** and all **Processors** reference the same **ProcessContainer**. This balances the processes evenly over the processors by assigning each idle processor a process from the pool of all ready processes. Such a **ProcessContainer** must provide mutual exclusion on its internal data structures since it will require concurrent accesses by multiple processors. As the number of processors increases, the number of invocations of *add* and *remove* will increase proportionally. The mutual exclusion overheads may, therefore, lead to a bottleneck. In order to decrease contention and to introduce process/processor affinity, the ready-queue function may be distributed between multiple **ProcessContainers** each assigned to a subset of the available processors. In order to balance the load, **Processes** can be migrated between these **ProcessContainers** when processors are idle or overloaded.

Another use of multiple ready-queues in *Renaissance* is to support multiprocessor gang scheduling in the form the Encore UMAX[Enc86] operating system provides. If a set of processors is assigned a **ProcessContainer** separate from the normal system ready-queue, those processors can be dedicated exclusively to processes placed in that container. For example, an application needing *exclusive* access to a set of processors can be supported in this way. The remaining processors could execute normal jobs without interference to the dedicated processors.

## 2.6 Deadlock and Race Avoidance

Multiprocessors introduce unique scheduling problems. In particular, when a **Process** is added to a **ProcessContainer**, it can potentially be removed immediately and run on another processor. The ready-queue is the best example of such a container. However, since no assumptions about the relative speeds of processes can be made on a multiprocessor, it is possible that a semaphore queue could exhibit the same problem if one process signals a semaphore simultaneously with another process waiting on it. Therefore, if a process adds *itself* to a container, there is a potential that it may begin simultaneously executing on two processors. Even though this will only be the case for the very short period of time until the first processor begins running another process, it could be disastrous as both processors will execute with the same stack, each overwriting the other's values.

The solution to this problem is one case where *Renaissance* differs completely from its predecessor, *Choices*. In *Choices*, a process is constrained to never add the **Process** representing itself to a **ProcessContainer**[RJC88]. Instead, *Choices* avoids the race condition by using a **ContextSwitchResponsibility** object. A **ContextSwitchResponsibility** object delegates the function of adding a running **Process** to a **ProcessContainer** to the process selected to run next. When a process relinquishes the processor it sends the *giveProcessorTo* message to the corresponding **Process**. Before doing this, however, it as-

signs a **ContextSwitchResponsibility** to the process being given the processor. The **ContextSwitchResponsibility** class defines a simple function which will be executed by the process given the processor before it resumes where it itself relinquished the processor. This function can be considered part of the “cost” of giving the processor to that process. The **DefaultResponsibility** class is used when the process relinquishing the processor has no special needs. **DefaultResponsibility** simply sends the **ready** message to the previous process so that it can execute again when a processor becomes idle. Other **ContextSwitchResponsibility** classes place the **Process** in a blocked or wait queue (see Section 2.7.2 for an example).

*Renaissance* uses a much simpler solution to the problem. By taking advantage of a simple state variable associated with each process object, it can be determined whether or not a process removed from a **ProcessContainer** is actually still executing on another processor. The **stillRunning** message can be sent to a **Process** to determine this. While a removed process is still running, a process wishing to relinquish the processor to that process simply spins waiting. Since the only cases where this can happen are within the *Renaissance* kernel itself, it can be assured that the time spent spinning is minimized.

A possibility of indefinite postponement is avoided in both *Renaissance* and *Choices* by assuring that interrupts are disabled between the point just before a **Process** is removed from a ready-queue and the point that the currently executing process relinquishes the processor. Doing this avoids the possibility of an interrupt occurring before the processor is actually relinquished resulting in a delay in running the removed process.

### 2.6.1 Optimizing Context Switching

The cost of using multiple processes in an application could be prohibitive if the expense of synchronizing and switching between them is too large. Therefore, many operating systems are designed to support *lightweight processes*. This is in some respects as misnomer. The goal of these systems is to minimize the cost of using multiple processes by minimizing the cost of switching contexts between them. They should, therefore, be described as supporting *lightweight context switching*. In other words, the “weight” of the process itself is often not at issue, rather the expense of context switching between processes. Interrupt and real-time processing likewise require that the overhead of context switching between processes be minimized.

Lightweight context switching is usually implemented by having processes share as much state as possible with each other. This reduces the time to switch between them because common state does not need saving or restoration. It also has the added advantage of reducing the amount of memory dedicated to storing per-process information. The most commonly shared state is a process’s address space.

Lightweight context switching is addressed in *Renaissance* by having subclasses of **Process** and **ProcessorContext** redefine their context switching methods in a way corresponding to the kind of process. For example, the *Renaissance* kernel code as implemented in the current prototype uses no floating point arithmetic. Therefore, it does not use the floating point registers. However, applications might require the floating point registers. Saving these registers when a system or interrupt process relinquishes the processor and restoring them upon its restart would be wasteful since their values are irrelevant. Therefore, only the particular subclass of **ProcessorContext** for **ApplicationProcesses** redefines **checkpoint** and **restore** to save and restore floating point registers.

This exemplifies an advantage of object-oriented programming when applied to operat-

ing systems. Important primitives like `giveProcessorTo` can be transparently optimized by redefining methods that they in turn rely on. Another advantage applies to easing operating system portability. When initially retargeting for a new architecture, it is easiest to implement the `checkpoint` and `restore` methods in the parent `ProcessorContext` class for that architecture in such a way as to save and restore the entire context of the processor. The subclasses for various kinds of processes can then inherit these methods. Optimizations like the one in the previous paragraph can be added later by redefining the methods in the subclasses. This is a specific case of specialization by subclassing.

### 2.6.2 Context Switching Performance

The context switching times between two system processes executing in the same address space was estimated to be approximately  $400\mu s$ . This number was calculated by implementing a pair of processes which continually relinquish the processor in a tight loop. This experiment was performed on an otherwise idle Encore Multimax with NS32332[Nat86] processors running at 15MHz. The system was run with a single processor to assure that the processes gave the processor back and forth to each other and did not execute in parallel. If one of the processes is changed to an application process, the time increases to around  $475\text{--}500\mu s$ . This extra expense is due to the loading and unloading of floating point registers and the expense of managing the application stack. If a pair of application processes executing in the same address space are used in this same experiment, the time increases to around  $550\mu s$ . The time to switch address spaces was estimated to be approximately  $168\mu s$ , by repeating this experiment with two application processes executing in different address spaces.

## 2.7 Mutual Exclusion and Synchronization

One of the biggest problems with programming parallel software is synchronization and providing mutually exclusive access to critical data. Object-oriented techniques have many advantages when applied to parallel and concurrent systems. Since the object-oriented principle of encapsulation allows only the methods of an object access to the state of the object, an object can control the exclusivity of accesses to its state through its methods. Thus an object can provide a "safe" interface to its instance variables. This idea is similar to Hoare's monitors[Hoa74], except that monitors guarantee mutually exclusive access only and preclude potentially concurrent access. Concurrent accesses to an object's data might be desirable in a multiprocessor or multiprogrammed operating system to increase performance. In object-oriented programming, the implementor of a class is free to code arbitrary restrictions on the ordering and mutual exclusivity of methods using semaphores, locks, or other similar techniques.<sup>5</sup>

*Renaissance* provides spin-locks (implemented by the `SpinLock` class) for low level mutual exclusion and semaphores (implemented by the `Semaphore` class) for both mutual exclusion and synchronization. These classes are used to implement protected data access.

---

<sup>5</sup>Although no current object-oriented languages currently support it, a technique like Campbell's path expressions[CH74] applied to an object-oriented languages could simplify the problem of reliably coding mutual exclusion between and within an set of messages. Complex requirements can be easily specified and implemented by synchronization at method entrance and exit. This type of ordering is very difficult to enforce in traditional programming paradigms. Usually, one is forced to explicitly code complex locking protocols directly into the invoking routines.

### 2.7.1 SpinLocks

Spin locks are provided for lightweight mutual exclusion by the **SpinLock** class as defined below:

```
class SpinLock : public Object {
protected:
    BusyWait busy;
    int previousInterruptState;
    Process * holdingProcess;
public:
    void acquire();
    void release();
};
```

The implementation of **SpinLock** relies on a small machine dependent class **BusyWait** which implements a busy-wait loop using whatever atomic test-and-set operations the target machine provides. The implementation of the **SpinLock** class assumes that the processor will not be relinquished while the lock is held. This assumption is checked at each context switch and a panic ensues if it is violated. The **acquire** message is sent to a **SpinLock** to enter a critical section. The corresponding method simply disables interrupts, and uses a test-and-set loop to wait for the **SpinLock** to be free:

```
SpinLock::acquire()
{
    int interruptState = thisProcess()->disableInterrupts();
    // Do the machine specific test-and-set.
    busy.wait();
    // Now (and only now) that the lock is held, the instance
    // variables can be updated.
    previousInterruptState = interruptState;
    holdingProcess = thisProcess();
    // Log the fact that the current process now holds a lock so
    // the context switching code can check.
    thisProcess()->incrementLocksHeld();
}
```

The **release** message is sent to a **SpinLock** to indicate that the **SpinLock** is free. The corresponding method releases the lock and re-enables interrupts if they were enabled when the **SpinLock** was first acquired:

```
SpinLock::release()
{
    // Access this variable and save its contents before the lock
    // is released because another process might immediately acquire it.
    int interruptsWereEnabled = previousInterruptState;
    holdingProcess = 0;
    thisProcess()->decrementLocksHeld();
    // Release the next process at the busy wait loop (if there is one)
    busy.releaseNextWaiter();
    // Re-enable interrupts if they were enabled when the lock was
    // first acquired.
}
```



```

        if( interruptsWereEnabled ) {
            thisProcess()->enableInterrupts();
        }
    }
}

```

Since there can only be one processor competing for a **SpinLock** at a time on a uniprocessor computer, versions of *Renaissance* for such computers can implement the **SpinLock** **acquire** method solely by disabling interrupts and **release** by restoring interrupts (if they were enabled when **acquire** was sent).

The time to acquire a **SpinLock** on the Encore Multimax is  $26\mu s$ . The cost of releasing the lock is  $18\mu s$ .<sup>6</sup> The vast majority of this time ( $16.8\mu s$ ) is spent obtaining a reference to the currently executing process. This function is not expanded in-line and obviously needs further optimization.

### 2.7.2 Semaphores

A semaphore is implemented by the **Semaphore** class and its **P** and **V** methods. The definition of the **Semaphore** class is shown below:

```

class Semaphore : public Object {
protected:
    BusyWait mutex;
    int count;
    ProcessContainer * queue;
public:
    Semaphore( int initialCount );
    ~Semaphore();

    virtual void P();
    virtual void V();
};

```

This class is a refinement of the *Choices* semaphore class introduced in [Rus91]. The *Renaissance* **Semaphore** class differs from that in *Choices* by its use of the simplified process scheduling primitives available in *Renaissance*.

As shown below, the implementation of the **P** method first disables interrupts then acquires exclusive access to the semaphore count by sending the **wait** message to the **mutex** instance variable (a **BusyWait**).

```

Semaphore::P()
{
    int wasInterruptable = thisProcess()->becomeUninterruptable();
    mutex.wait();
    count--;
    if( count < 0 ) {
        queue->add( thisProcess() );
        mutex.releaseNextWaiter();
        thisProcess()->block();
    }
}

```

<sup>6</sup>It should be noted that the C++ compiler does in-line expansion of the **acquire** and **release** methods for efficiency.

```

    }
    else {
        mutex.releaseNextWaiter();
    }
    if( wasInterruptable ) thisProcess()->becomeInterruptable();
}

```

The P method decrements the count and tests to see if the invoking process must block. If the count is greater than or equal to zero the invoking process can continue. This is accomplished by sending **release** to the **mutex** instance variable to release mutually exclusive access on the semaphore count and then re-enabling interrupts and returning.

If the count was negative the invoking process must block. In retrospect, this is where the *Choices* restriction of a process not being able to add itself to the semaphore's wait queue (see Section 2.6) is most awkward. In *Choices*, the process is added to the queue by the process being *given* the processor by using a **ContextSwitchResponsibility**. First **getNextReadyProcess** is sent to the current **Processor** in order to get another process to run. Then, that **Process**'s responsibility is set to **SemaphoreResponsibility**. Finally, the P method causes the processor to be relinquished and the next process is run by sending **giveProcessorTo** to the current **Process**:

```

ChoicesSemaphore::P()
{
    int wasInterruptable = thisProcess()->becomeUninterruptable();
    mutex.acquire();
    count = count - 1;
    if( count < 0 ) {
        Process * nextProcess = thisProcessor()->getNextReadyProcess();
        nextProcess->setResponsibility( SemaphoreResponsibility, this );
        thisProcess()->giveProcessorTo( nextProcess );
    }
    else {
        mutex.release();
    }
    if( wasInterruptable ) thisProcess()->becomeInterruptable();
}

```

As shown below, the implementation of **SemaphoreResponsibility** simply places the blocking process in the semaphore's wait queue:

```

SemaphoreResponsibility( Process * oldProcess, Semaphore * mySemaphore )
{
    // mySemaphore is an instance variable set when the
    // SemaphoreResponsibility is created. It references
    // the associated semaphore.
    mySemaphore->queue->add( oldProcess );
    mySemaphore->mutex.release();
}

```

It should be apparent that the implementation of the P method in *Renaissance* is decidedly simpler. Information about the ready-queue is completely removed along with the need for a **SemaphoreResponsibility** to place the blocking process in the semaphore's

block queue. The process does this directly and effects a context switch to another process by sending itself the block message.

The V method is implemented by acquiring mutually exclusive access to the semaphore count, incrementing the count, and testing if the count is still nonpositive. If the count is positive, then no other processes are blocked and the invoking process can continue by releasing the mutual exclusion on the count, re-enabling interrupts, and returning. If there is a process waiting, then it is first removed from the **Semaphore's** block queue and sent the ready message.

```
Semaphore::V()
{
    int wasInterruptable = thisProcess()->becomeUninterruptable();
    mutex.acquire();
    count = count + 1;
    if( count <= 0 ) {
        Process * waiter = queue->remove();
        mutex.releaseNextWaiter();
        waiter->ready();
    }
    else {
        mutex.releaseNextWaiter();
    }
    if( wasInterruptable ) process->becomeInterruptable();
}
```

### 2.7.3 Alternate Semaphore Implementations

*Renaissance* uses the **GraciousSemaphore**[Rus91] subclass of **Semaphore**, introduced originally in *Choices*, to implement a form of semaphore that causes the current process to immediately relinquish the processor when the V message is sent and there are blocked processes. This class is useful in implementing processes that block awaiting an interrupt, and then should run as soon as the interrupt occurs. The interrupt handler simply has to send the V message to a **GraciousSemaphore** upon which the waiting process is blocked. The process which is executing when the message is sent suspends itself by sending the **giveProcessorTo** message to the **Process** object corresponding to itself, with the waiting processes as an argument. The implementation of the **GraciousSemaphore** V method is shown below. The P method is inherited from the parent **Semaphore** class without change.

```
GraciousSemaphore::V()
{
    int wasInterruptable = thisProcess()->becomeUninterruptable();
    mutex.acquire();
    count++;
    if( count <= 0 ) {
        Process * waiter = queue->remove();
        mutex.releaseNextWaiter();
        thisProcess()->giveProcessorTo( waiter );
    }
    else {
        mutex.releaseNextWaiter();
    }
    if( wasInterruptable ) thisProcess()->becomeInterruptable();
}
```

Acquiring a non-busy semaphore on the Encore Multimax takes  $88.4\mu s$ . Releasing a semaphore takes  $85.3\mu s$ . These data were acquired by first sending the P message to a **Semaphore** with a positive count, and then sending the V message. Therefore, these numbers reflect the cost of acquiring a free **Semaphore** and releasing a **Semaphore** on which no other processes are blocked. If a **Semaphore** is not free when the P message is sent to it, then the time before the P method returns depends on the length of time until another process sends the V message to the **Semaphore**. If another process were blocked awaiting the **Semaphore** when the V message was sent, then the time would increase by the amount of time necessary to dequeue the waiting process and send it the **ready** message.

To give an estimate of the overhead of using semaphores for synchronization, a test was run where two system processes looped exchanging the processor by alternately sending the P and V messages to a pair of semaphores. The first process sent the V message to one semaphore then the P message to another, while the second process did the opposite. A single iteration of the loop took approximately  $1000\mu s$ . As discussed earlier, a context switch between a pair of system processes takes approximately  $400\mu s$ . Two context switches occur for each iteration of this loop. The first when the process blocks on the semaphore, and the second when the processor is reacquired when the other process relinquishes it. Therefore, subtracting the cost of the two context switches leaves an overhead of approximately  $200\mu s$ . Based on the data above, this is approximately the expected cost of a P followed by a V.

### 3 C++ as an Implementation Language

The programming language used to implement an object-oriented operating system can drastically affect its performance. The implementation language chosen for both *Choices* and *Renaissance* is C++. This is mainly due to the performance advantages of statically typed object-oriented languages, along with the added advantage that, although they violate the pure object-oriented paradigm, C++ allows certain low-level programming techniques necessary for the easy and efficient implementation of an operating system. In particular, the language allows the programmer to specify an object's representation in memory, to place objects at a specific address, and to predetermine the size of an object. Specifying an object's representation in memory is necessary to allow classes to represent hardware-defined entities such as device or processor control registers and device command/control messages. It is also necessary in order to allow data structures specified by certain standards, such as the representation of a file on disk or the placement of fields in a network protocol packet, to be encapsulated within objects that are instances of representative classes. The ability to specify a new object's location in memory is necessary to allow the addressing of hardware specified entities as objects after representative classes have been designed. Again, this includes entities like device registers or hardware-defined data structures that are often at a fixed location in memory. Finally, the ability to precisely determine the size of an object is useful in optimizing memory allocation/deallocation for frequently instantiated classes.

C++ does not always faithfully implement the object-oriented paradigm. It can, however, be used as an object-oriented language and allows an examination of the advantages of object-oriented programming applied to operating systems. C++ supports objects, classes, inheritance and polymorphism. However, not every value in a C++ program is an object. This is a concession to simplicity of code generation and optimization since, in particular, primitive types such as integers, floating point numbers and characters are not objects that are instances of representative classes. Having such primitive types built into the language



allows the compiler to generate traditional code for operations on such types. Specifically, because no method lookup is done for such operations, straight one-to-one mappings to machine code exist and can be expanded in-line in the generated code. Another violation of the pure object-oriented programming is that C++ allows direct access to instance variables although this mechanism does not have to be used.

Perhaps the biggest drawback of C++ is that it implements object polymorphism solely based on class inheritance.<sup>7</sup> This is mostly a concession to efficiency and, along with static typing, allows C++ to implement a very fast method invocation scheme. However, it forces the programmer to constrain the type hierarchy to the class hierarchy, i.e., a concrete class implementing a desired signature *must* be a subclass of the abstract class defining the signature. Solutions to this are proposed in [RG91].

## 4 Conclusion

Low-level and architecture specific details of process management for an operating system must be efficiently implemented. *Renaissance* provides an excellent demonstration that object-oriented programming and design techniques can efficiently and simply support such details even in a multiprocessor environment.

The *Renaissance* process management provides an object-oriented interpretation and implementation of process scheduling, synchronization and context switching. Like *Choices* before it, *Renaissance* attempts to let the operating system optimize context switching based on the requirements of the processes themselves. This is accomplished by using an abstract class to define a process and then subclassing that class to implement the process abstraction with various performance enhancements. Unlike *Choices*, *Renaissance* allows scheduling of processes under application control and provides a simpler, cleaner model of process context switching.

The *Renaissance* `giveProcessorTo` primitive provides the mechanism to effect context switching between processes. Policy decisions are implemented by different scheduling and synchronization objects, for example, **Semaphores**. In many cases, inheritance can be used to alter policy decisions. For example, the **GraciousSemaphore** class redefines the `V` method of **Semaphore** to implement a different policy when a process blocked on a semaphore is resumed. Even a **Semaphore** may have different policies. For example, the queuing in the **Semaphore** implementation relies on the interface **ProcessContainer** provides. Various classes implementing the **ProcessContainer** signature can be used to alter the behavior of a **Semaphore**. For example, the **ProcessContainer** representing the queue of blocked processes may implement a FIFO or priority-based scheme.

In summary, *Renaissance* provides a flexible, efficient, and object-oriented interpretation of process context switching, scheduling, and exception handling.

## 5 Availability

The initial prototype of the *Renaissance* system is currently being completed. Remaining are the implementations of a redesigned virtual memory management system and file system. The implementation of the distributed object messaging facility is also underway.

---

<sup>7</sup>To get even this behavior, methods used in argument classes must be implemented as C++ *virtual functions*[Str86].

Likewise, ports to the SPARC and Hewlett-Packard Precision architectures have been undertaken. When complete, the system will be distributed with as little restriction on use and redistribution as possible. Interested parties should contact the author.

## References

- [CH74] R. H. Campbell and A. N. Habermann. The Specification of Process Synchronization by Path Expressions. In G. Goos and J. Hartmanis, editors, *Operating Systems, International Symposium, Rocquencourt*, volume 16 of *Lecture Notes in Computer Science*, pages 89–102. Springer-Verlag, New York, April 1974.
- [Dei84] Harvey M. Deitel. *An Introduction to Operating Systems*. Addison-Wesley Publishing Company, Reading, Massachusetts, Reading, Massachusetts, 1984.
- [Dij68] Edsger W. Dijkstra. The Structure of the THE Multiprogramming System. *Communications of the ACM*, pages 341–346, May 1968.
- [Enc86] Encore Computer Corporation, Marlboro, Massachusetts. *UMAX 4.2 Programmer's Reference Manual*, 1986.
- [Hoa74] C. A. R. Hoare. Monitors, An Operating System Structuring Concept. *Communications of the ACM*, October 1974.
- [Nat86] National Semiconductor Corporation, Santa Clara, California. *Series 32000 Databook*, 1986.
- [PS85] James L. Peterson and Abraham Silberschatz. *Operating System Concepts*. Addison-Wesley Publishing Company, Reading, Massachusetts, second edition, 1985.
- [RC89] Vincent Russo and Roy Campbell. Virtual Memory and Backing Storage Management in Multiprocessor Operating Systems Using Object-Oriented Design Techniques. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, 1989.
- [RG91] Vincent F. Russo and Elana D. Granston. Signature Based Polymorphism for C++. In *Proceedings of the USENIX C++ Conference*, 1991.
- [RJC88] Vincent Russo, Gary Johnston, and Roy Campbell. Process Management and Exception Handling in Multiprocessor Operating Systems Using Object-Oriented Design Techniques. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, 1988. Also Technical Report No. UIUCDCS-R-88-1415, Department of Computer Science, University of Illinois at Urbana-Champaign.
- [Rus91] Vincent F. Russo. *An Object-Oriented Operating System*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1991.
- [Str86] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

# A Hybrid Approach to Load Balancing in Distributed Systems

Prabha Gopinath  
pbg@philabs.Philips.com  
Philips Laboratories  
North American Philips Corp.  
345 Scarborough Road  
Briarcliff Manor, NY 10510

Rajiv Gupta  
gupta@cs.pitt.edu  
Univ. of Pittsburgh  
Dept. of Computer Science  
Pittsburgh, PA 15260

**Abstract** - The processors in a distributed system can be viewed as being in a lightly loaded, heavily loaded, or normally loaded state. The goal of load balancing algorithms is to keep all nodes in a normally loaded state by migrating processes from heavily loaded nodes to lightly loaded nodes. In addition, load balancing must involve low communication overhead and should respond quickly to load imbalances in the system. Traditional load balancing strategies can be classified into two broad categories, namely sender initiated and receiver initiated. In this paper we present a hybrid algorithm for performing dynamic load balancing in a distributed system. The system is partitioned into disjoint groups of processors. First intra-partition load balancing is performed using a receiver initiated strategy to achieve an acceptable load distribution. If this is not sufficient, inter-partition load balancing is carried out using a sender initiated strategy. Simulation results demonstrate that the hybrid approach compares favorably with the drafting and bidding load balancing algorithms for a point-to-point linked network.

**Keywords** - load balancing, distributed systems, process migration, drafting, bidding, partitioning.

## 1. Introduction

For our purposes a distributed system consists of a collection of autonomous processors, each with its own operating system, which communicate via message passing over communication links. There is no shared memory and message communication takes nonnegligible time. Depending upon the topology of the system a message from one processor to another may have to pass through intermediate nodes. To achieve high performance in a distributed system, it is essential to balance the load of various processors. Dynamic load balancing can be achieved by migrating processes from heavily loaded processors to lightly loaded processors in the system. For this reason process migration is supported by several distributed systems including Demos [2] and Emerald [1]. The two main desirable characteristics of a load balancing protocol are **low communication overhead** and a **fast response time** to load imbalance. Low communication overhead is desired because excess message traffic may further increase the load on already heavily loaded processors. Fast response time is essential so that processes from a heavily loaded processor are migrated to a lightly loaded processor while the latter is still in a lightly loaded state.

Load balancing strategies can be broadly classified into two main categories, namely sender initiated and receiver initiated. In sender initiated strategies a heavily loaded processor, i.e., the processor from which processes have to be migrated, initiates the load balancing protocol. The **bidding** [3] algorithm is an example of a sender initiated load balancing strategy. In receiver initiated strategies lightly loaded processors, i.e., receivers of migrated processes, initiate the load balancing protocol. The **drafting** [4] algorithm is a receiver initiated algorithm. In this paper we present a hybrid load balancing strategy that improves upon the response time and communication overhead characteristics of both the bidding and drafting algorithms. The hybrid algorithm is a combination of sender initiated and receiver initiated strategies.

In the **bidding** algorithm a heavily loaded processor wishing to reduce its load by migrating a process sends out a request for bids to all other processors in the system. After all bids have been received, they are evaluated and the process is migrated to the processor corresponding to the best bid. A drawback of this algorithm is that it is the heavily loaded processors that must send out requests for bids and evaluate the replies. This further increases their load. Another disadvantage is that all the heavily loaded processors

use the same algorithm to evaluate bids and therefore pick the same lightly loaded processor to which migrate. This processor may get overloaded if several senders select its bids. Thus, the protocol may suffer from overmigration of processes and subsequent thrashing. To avoid this problem the bidder may either accept or reject a migrated process. If the process is rejected, the bidding procedure starts over again. However, repeated execution of the bidding procedure is undesirable because it results in a great deal of communication overhead and further overloads an already heavily loaded processor. Furthermore, rejection of processes leads to delays in process migration and hence in achieving a balanced load distribution.

In the **drafting** algorithm lightly loaded processors send out draft requests looking for additional work. Upon receiving replies, a lightly loaded processor evaluates and chooses the best reply and then drafts a process from that processor. Unlike bidding, the heavily loaded processors are not burdened with executing the drafting procedure. However, a heavily loaded processor replies to all draft requests which it receives. Thus, during drafting the communication overhead on the heavily loaded processor increases with the number of lightly loaded processors in the system. Furthermore, the most heavily loaded processor is likely to be picked by several lightly loaded processors since it responds to multiple draft requests. This will cause a heavily loaded processor to become lightly loaded. Thus, as in bidding, this approach may also suffer from overmigration of processes. To avoid overmigration, a drafted processor can either accept or reject an offer to migrate a process to the drafting processor. If the draft is rejected, the drafting procedure starts over again. However, repeated execution of the drafting procedure consumes system communication bandwidth, slows down the system's response to load imbalances, and may cause communication overheads on the heavily loaded nodes.

Consider the situation in which the system contains several lightly loaded and several heavily loaded processors. In the bidding algorithm several of the heavily loaded processors will choose the most lightly loaded processor. However, this lightly loaded processor may not be able to accept processes from all heavily loaded processors. Thus, some of them will have to repeat the bidding procedure. Thus, at a given point in time only some of the heavily loaded processors may be able to migrate processes even if there are several lightly loaded processors in the system. In the drafting algorithm several lightly loaded processors will draft processes from the most heavily loaded processor. Only some of the lightly loaded processors are likely to receive a process from the heavily loaded processor. The remaining lightly loaded processors must repeat the drafting procedure. Thus, at a given point in time, the drafting procedure tends to reduce the load of only one of the heavily loaded processors. However, to ensure fast response to load imbalance in a system with several heavily loaded processors and several lightly loaded processors, it is essential that different lightly loaded processors simultaneously receive processes from different heavily loaded processors.

The communication overhead will be very high if the draft/bid requests are sent to all processors in the system. To control the overhead, draft/bid messages are sent only to those processors that are within a certain number of hops, say  $h$ . The disadvantage of this approach is that process migration will not occur if the lightly loaded processors and heavily loaded processors are separated by more than  $h$  hops. Thus, load balancing may not occur even if the system has several lightly loaded processors and several heavily loaded processors. Since the protocols for bidding and drafting create a fair amount of communication overhead it is desirable to migrate processes in groups so as to amortize this overhead.

In this paper, we present a **hybrid** load balancing algorithm that avoids the drawbacks of bidding and drafting described above. A distributed processor system is partitioned into disjoint groups of processors (see Fig. 1). A modified form of drafting, which avoids overmigration and the related repeated execution of the drafting procedure, is used to perform load balancing within each partition. The modifications to the drafting algorithm result in better response times. The partitioning is done in a manner such that the heavily loaded links connect different partitions. Since the messages generated by the drafting protocol are unlikely to cross partition boundaries, the drafting messages will not increase the communication load on the already heavily loaded links. If a partition contains no lightly loaded nodes, then to reduce the load of the heavily loaded processors in that partition, inter-partition process migration is carried out. The inter-partition process migration procedure is executed by the heavily loaded nodes, which makes it similar to the bidding procedure. However, this protocol is much simpler and requires much fewer messages than the bidding procedure. Low communication overhead is especially desirable in this case since the messages will travel among the partitions using heavily loaded links. Since inter-partition migration is more



expensive than intra-partition migration, the former is initiated infrequently and only in situations when the latter fails to balance the load. Drafting is more suitable for intra-partition load balancing because a lightly loaded node uses load tables to send out load requests. The load tables can be kept updated only if the nodes can communicate their load states quickly. If the communication time is high, by the time a processor receives the new state of a sender the sender's state may already have changed again.

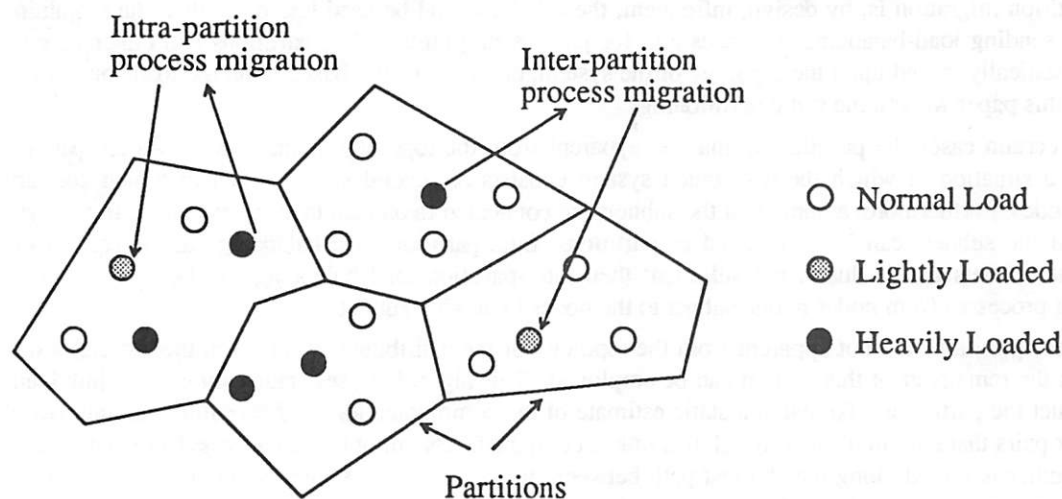


Figure 1. Hybrid Load Balancing Algorithm

## 2. Related Work

In addition to bidding and drafting there are two other algorithms that have been studied. These are the random [5] and the gradient [6] strategies. In the random strategy, proposed by Eager *et al* [5], the sender randomly selects the destination for a process to be migrated. It has been demonstrated that the random strategy performs well in comparison to more complex strategies such as bidding and drafting. This is because the bidding and drafting strategies are so complex that by the time a decision to migrate a process is taken the loads at the sending and receiving processors may have significantly changed. To avoid this problem the hybrid algorithm proposed in this paper employs extremely simple forms of sender initiated and receiver initiated strategies. If the system contains too many heavily loaded nodes the random strategy will further slow down the system by unnecessarily migrating processes from one heavily loaded node to another. In the gradient [6] method migrations are continuously performed between immediate neighbors so as to maintain an even load distribution. The major drawback of this strategy is thrashing. If the loads of two neighboring nodes fluctuate in such a fashion that a different node is more heavily loaded after each fluctuation in load then processes will continue to be migrated back and forth between the two nodes.

A hybrid algorithm for load balancing has also been proposed by Chowkwanyun and Hwang [8]. In their approach they describe a technique whereby the nodes in a distributed system dynamically switch between sender-initiated and receiver-initiated modes of operation. The receiver initiated mode is based on the drafting protocol described earlier. The sender initiated mode, however, is based on the gradient strategy. This suffers from the problems of over-migration and thrashing described above. In addition this hybrid technique does not partition the system. Rather load balancing is performed only for nodes that are within a certain "network diameter"  $\Delta$ , or number of hops from a given processor. The disadvantage of this approach, as we have pointed out earlier, is that load balancing is artificially restricted and will not occur if heavily loaded and lightly loaded processors are separated by a distance greater than  $\Delta$ .

In the next section we discuss the static partitioning of a system for the hybrid algorithm. Following this the intra-partition and inter-partition process migration algorithms are discussed in detail. The bidding, drafting and the hybrid algorithms have been implemented and their performance studied on a simulated distributed system. The simulation results comparing the response time and communication overhead of these algorithms are presented.

### 3. Partitioning a Distributed System for Load Balancing

We now consider the problem of partitioning a distributed system for the purpose of load balancing. Ideally the partitions should be constructed in such a manner that the links that would be likely to be used more frequently for sending application messages connect nodes belonging to different partitions. Since inter-partition migration is, by design, infrequent, these links would be used less often than the remaining links for sending load-balancing messages and for process migrations. The partitions can either be constructed statically, based upon the topology of the system, or dynamically, based upon the loads on various links. In this paper we assume static partitioning.

In certain cases the partitioning may be apparent from the topology of the system. As an example consider a situation in which the distributed system consists of several subnets each of which contains several nodes. Furthermore, assume that the subnets are connected to one another via gateways. In this type of system the subnets can be considered as partitions. Intra-partition load balancing can be carried out within the subnets and if this is not sufficient then inter-partition load balancing can be carried out by migrating processes from nodes in one subnet to the nodes in another subnet.

If the partitions are not apparent from the topology of the distributed system then the algorithm discussed in the remainder of this section can be employed. This algorithm uses static estimates of link loads to construct the partitions. To obtain a static estimate of the communication load for a link, the number of processor pairs that communicate through this link is computed. Assuming that a message from one processor to another is routed along the shortest path between the two processors, the communication load for a link can be expected to be high if the number of processor pairs for which the shortest path includes the link is high. Given the topology of the system, this number can be statically computed for each link.

The construction of partitions is started by choosing the link with the highest expected load. The processors connected by this link are put in separate partitions. Additional nodes are added to a newly created partition, one at a time, till it is of a desired size. To choose the next node to be included in a partition, all nodes directly connected to the partition are examined. Using a heuristic criteria one of the nodes is chosen. Two possible heuristics for choosing a node are:

- (i) For each node  $n_i$ , connected to the partition, compute the sum of *Expected-load's* of all the links that connect  $n_i$  to a node in the partition. Choose the node for which this sum is the minimum; or
- (ii) For each node  $n_i$ , connected to the partition, find the maximum of *Expected-load's* of all the links that connect  $n_i$  to a node in the partition. Choose the node for which this maximum value is the smallest.

If several nodes satisfy the above criteria a single node from these nodes can be chosen as follows. A node which is directly connected to the maximum number of nodes already in the partition can be selected. After a partition is of the desired size or no more nodes can be added to it, new partitions are started and the above process is repeated till the entire system has been partitioned. If partitions smaller than a desired size result, they can be merged with one of the neighboring partitions. The algorithm is as follows:

---

#### Algorithm for Static Partitioning

```
{
  for each link  $l$  compute the following:
     $Expected-Load(l) = \#$  of shortest paths that contain  $l$ 
  Sort the links in decreasing order of their Expected-Load values
  loop {
    From among the links that connect two processors
    of which at least one is yet to be included in a partition
    select the link  $l=(p_1, p_2)$  that has the
      maximum Expected-Load value
    if  $p_1$  does not belong to any partition then
      create new partition =  $\{p_1\}$ 
    if  $p_2$  does not belong to any partition then
```

```

        create new partition = {p2}
        continue to add elements to partitions, giving
        preference to those that are connected by links
        with low Expected-Load values, till they are of
        desired size or no more nodes can be added
    } until all partitions have been constructed
    if a partition is smaller than some threshold size then
    merge it with the smallest neighboring partition.
}

```

An example demonstrating the above process is shown in Fig. 2. In this example a large number of processors must use the link between processors *p1* and *p2* to communicate. As a result *p1* and *p2* must be in different partitions. Assuming that each partition is allowed to contain up to six nodes the system will be divided into two partitions as shown.

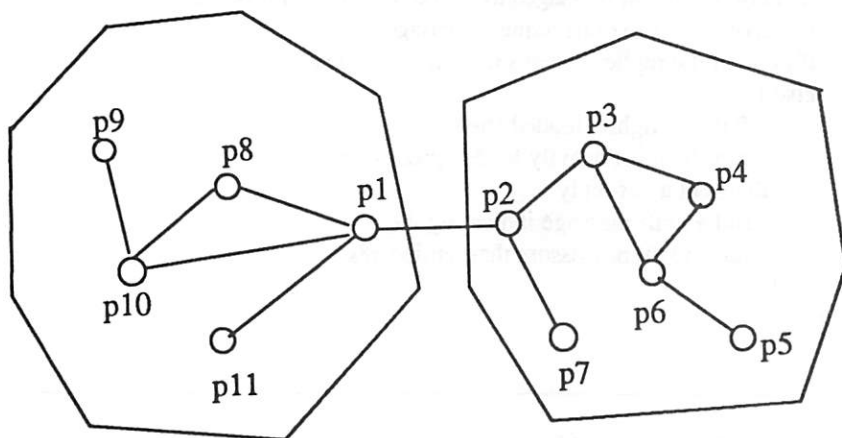


Figure 2. Partitioning a Distributed System

Another approach to the above problem is to construct and modify the partitions based upon the actual run-time loads on the links. A dynamic strategy would require measuring link loads at run-time.

#### 4. Intra-partition Process Migration

Intra-partition load balancing is performed using a modified form of drafting. Each processor maintains a load table which indicates whether each of the other processors in the partition is in a lightly loaded, normally loaded, or heavily loaded state. A processor communicates a change in its load state to other processors by sending explicit *load table update* messages. Alternative strategies for updating load tables, varying from broadcast to selective updating, are described in [4].

If a processor is lightly loaded, and its load table indicates that there are some heavily loaded processors in the partition, the drafting process is triggered. The processor sends *draft request* messages to all the heavily loaded processors in the partition. The receiving processors respond to the draft request message with either a *no* message, or a *maybe* message, or a *yes* message. In addition, they send their current loads as part of the message. A processor responds with a *no* message if it is no longer heavily loaded. A *yes* message is sent if the processor is willing to be drafted and the number of *yes* responses it has already sent out is less than the number of processes it needs to migrate to return to a normally loaded state. A *maybe* message is sent if the processor has already sent out sufficient *yes* messages.

After receiving replies from all processors, the lightly loaded processor drafts the processor that is most heavily loaded and has responded with a *yes* message to the draft request. A message indicating the choice is sent to all heavily loaded processors that responded with a *yes* message. Upon receiving this message the drafted processor, if still heavily loaded, migrates a process to the drafting processor. If the

drafting processor receives a *load table update* message indicating that the drafted processor is now lightly loaded, prior to receiving a process from the drafted processor, it knows that it will not receive a process. A processor that has responded with a *yes* message, but has been not yet been drafted, is free to respond to an additional draft request with a *yes* message. If the draft requests by a lightly loaded processor yield *maybe* responses, but no *yes* responses, the processor repeats the drafting procedure at a later time. If only *no* responses are received the processor must wait till its load table indicates the presence of heavily loaded nodes.

---

#### Drafting Algorithm Executed by Processor *P*

```
{
  if processor P is lightly loaded and its load table indicates
  that there are heavily loaded processors in its partition
  {
    Send draft request messages to the heavily loaded processors
    Receive replies to draft request messages
    if none of the replies is a yes message retry later
    else {
      if P is still lightly loaded then
        Draft the most heavily loaded processor
        that sent a yes reply
        Send a draft message indicating the
        choice to all processors that replied yes
      }
    }
  }
```

---

#### Respond to a Draft Request Message

```
{
  if not heavily loaded then message-type = no
  else {
    Based on the current load compute num-overload,
    the number of processes that should be migrated
    to bring the state of the node from heavily loaded
    to normally loaded

    if  $\text{num-overload} \leq \text{num-pending}$ 
    then message-type = maybe
    else {
       $\text{num-pending} = \text{num-pending} + 1$ 
      message-type = yes
    }

    Send a reply message of message-type and current
    processor load to the drafting processor
  }
}
```

---



---

**Received a Migrate Process Message**

```
{  
  if drafted-processor == self {  
    if still heavily loaded then migrate a process  
    else reject offer  
  }  
  else if a yes message had been sent to the drafting  
    processor then num-pending = num-pending - 1  
}
```

---

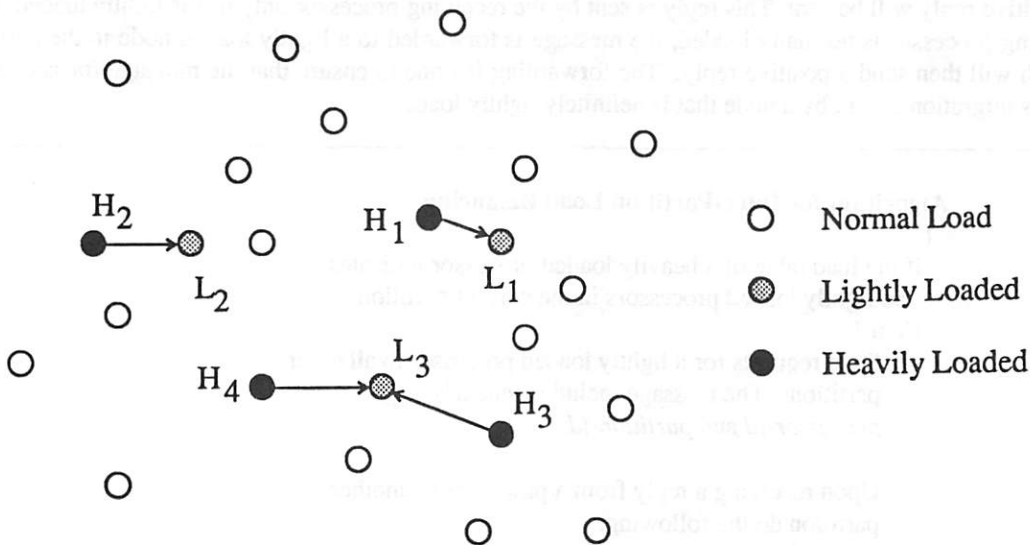


Figure 3. Intra-partition Drafting

The drafting algorithm presented in this section reduces the likelihood of having to repeat the drafting process. Consider the scenario shown in Fig. 3 which represents one partition in a system. The partition contains three lightly loaded processors ( $L_1$ ,  $L_2$ , and  $L_3$ ) and four heavily loaded processors ( $H_1$ ,  $H_2$ ,  $H_3$ , and  $H_4$ ). All three lightly loaded processors simultaneously execute the drafting procedure. Let us assume that each of the heavily loaded processors wishes to migrate a single process. The lightly loaded processors will receive *yes* replies from different heavily loaded processors (for example,  $L_1$  from  $H_1$ ,  $L_2$  from  $H_2$ , and  $L_3$  from  $H_3$ ,  $H_4$ ). Thus, process migration of processes to all three lightly loaded processors can begin simultaneously. Since each heavily loaded processor will only reply to a single drafting message with a *yes* message, all three lightly loaded processors will never draft the same heavily loaded processor.

Next we analyze the message complexity of the intra-partition load balancing algorithm. For simplicity we assume that the load tables of all nodes accurately reflect the state of the system. Let  $L$  denote the number of lightly loaded processors and  $H$  the number of heavily loaded processors in a partition. Draft requests are sent by each lightly loaded processor to each heavily loaded processor. Thus,  $L \cdot H$  draft request messages are sent out and an equal number of replies are sent back. Let  $P$  denote the total number of processes that all the heavily loaded processors together must migrate to achieve a normally loaded state. Among the  $L \cdot H$  replies, the number of *yes* messages will be  $\text{minimum}(P, L \cdot H)$ . Thus, after choosing the processor to draft, the lightly loaded processors must send out  $\text{minimum}(P, L \cdot H)$  messages indicating their draft choices. Therefore, the total number of messages sent in this entire procedure is  $\text{minimum}(P, L \cdot H) + 2L \cdot H$ . This indicates that the number of messages sent increases with the number of heavily loaded processors and the degree of load imbalance. The above analysis does not include the messages sent to update load tables. It may be possible to piggyback some of the load table update messages onto the messages sent during the intra-partition process migration protocol.

## 5. Inter-partition Process Migration

If a partition contains no lightly loaded processors, the load of the heavily loaded processors cannot be reduced through intra-partition process migration as described in the previous section. In this situation process migration across partitions may result in a better load distribution. A heavily loaded processor initiates the inter-partition process migration protocol if it does not receive any draft requests and its load table indicates that there are no lightly loaded processors in the partition. Messages requesting inter-partition process migration are sent to all other partitions. A single message is sent to every partition and it is sent to the node in the receiving partition that is closest to the sender. This node can be found statically from the topology of the system. A processor, upon receiving such a request from another partition, examines its load table to determine if the partition contains any heavily loaded nodes. If heavily loaded nodes are present the message is discarded. On the other hand, if no heavily loaded nodes are present in the partition, a positive reply will be sent. This reply is sent by the receiving processor only if it is lightly loaded. If the receiving processor is normally loaded, the message is forwarded to a lightly loaded node in the partition, which will then send a positive reply. The forwarding is done to ensure that the message for accepting process migration is sent by a node that is definitely lightly loaded.

---

### Algorithm for Inter-Partition Load Balancing

```
{
  if the load table of a heavily loaded processor indicates
    no lightly loaded processors in the current partition
  then {
    Send requests for a lightly loaded processor to all other
    partitions. The message includes sender's
    processor-id and partition-id.

    Upon receiving a reply from a processor in another
    partition do the following:
    - if draft requests are received prior to receiving
      this reply ignore the message.
    - if processor is now lightly loaded ignore message.
    - if the above conditions are not true select a
      process to migrate and send it to the lightly
      loaded processor from another partition.
  }
}
```

---

The algorithm for inter-partition process migration is similar to bidding in that it is executed by a heavily loaded processor. However, it is much simpler than bidding. A heavily loaded processor simply migrates a process to the first processor that responds positively. It does not wait for bids from all lightly loaded processors in other partitions and does not have to evaluate all the bids to choose the best one. The number of messages sent is small since only one request message is sent to each partition. The task of responding is also simple as a reply is only sent if a processor is willing to receive a process, so the reply does not need to contain the sender's current load.

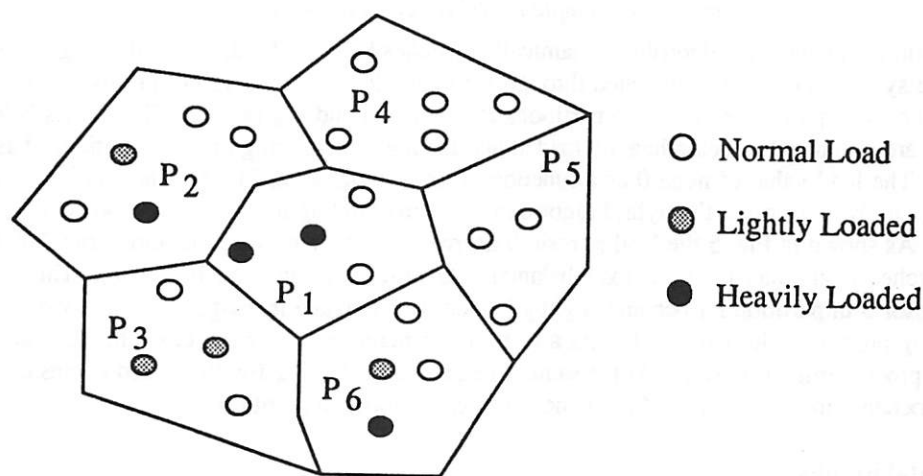
---

#### Respond to a Request Message

```
{
  if the request is for processors in another partition
  then propagate the message.

  if the request is meant for the current partition then
  {
    if the current partition contains heavily loaded
    processors or no lightly loaded processors ignore
    the message. The presence of lightly/heavily loaded
    processors is determined by examining the load table.
    else
    if lightly loaded respond to the sender asking
    for process migration.
    else
    if not lightly loaded forward the message to a
    lightly loaded processor in the partition.
  }
}
```

---



*Figure 4. Inter-partition Migration*

Fig. 4 shows a system with six partitions. Partition  $P_1$  contains two heavily loaded processors but no lightly loaded processors. The heavily loaded processors will therefore send out requests to other partitions to perform load balancing. Upon receiving these requests, partitions  $P_2$ ,  $P_4$ ,  $P_5$ , and  $P_6$ , will ignore them because they either contain heavily loaded processors or have no lightly loaded processors. However, the lightly loaded processors from partition  $P_3$  will respond positively, possibly causing the processors from partition  $P_1$  to migrate processes.

The message overhead of the inter-partition protocol is quite low. A heavily loaded processor executing this protocol sends one message to each of the other partitions. After such a message is received by a partition, in most cases it will be forwarded only once if the load tables are fairly accurate. The heavily loaded node from the sending partition receives at most one reply from each of the receiving partitions. Thus, if the partitions are large this number can be quite small.

## 5.1. Switching Between Bidding and Drafting

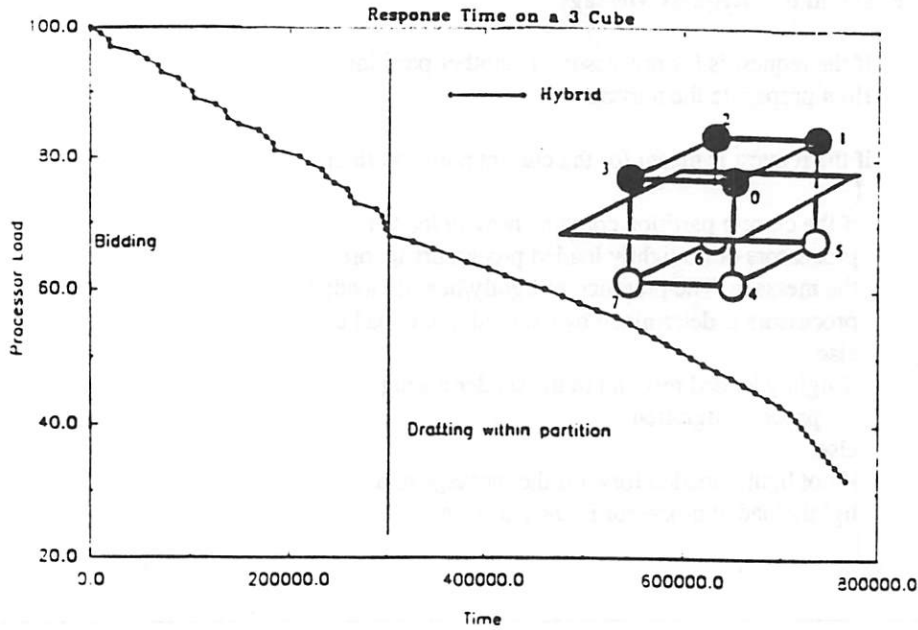


Figure 5. An Example of Hybrid Load Balancing

The hybrid load balancing algorithm dynamically switches between bidding and drafting based upon the state of the system. This is demonstrated through the example shown in Fig. 5. In this experiment an 8-node hypercube was partitioned into two partitions  $P_1=(0,1,2,3)$  and  $P_2=(4,5,6,7)$ . The nodes belonging to partition  $P_1$  are initialized as being heavily loaded and the nodes belonging to  $P_2$  are initialized as being lightly loaded. The load value of node 0 as a function of time is shown in Fig. 5. Since no intra-partition load balancing can be performed the hybrid algorithm performs bidding to reduce the loads at the heavily loaded nodes. As shown in Fig. 5 the load at node 0 decreases with time. At simulation time 300000 the algorithm switches from inter-partition load balancing to intra-partition load balancing. This happens because processor 3 in partition  $P_1$  becomes lightly loaded. Load update messages are sent by processor 3 to the remaining processors in partition  $P_1$ . As a result the remaining processors cease bidding and hence inter-partition process migration stops. At the same time processor 3 being lightly loaded begins executing the drafting procedure to reduce the load of the heavily loaded nodes in partition  $P_1$ .

## 6. Experimental Results

The performance of the hybrid algorithm was compared with the bidding and drafting algorithms through simulation studies. The LANSF protocol modeling environment [7] was used to carry out these experiments (Also see Appendix A.). LANSF allows specification of arbitrary topologies although the results presented in this section are based upon an 8-node hypercube. The partitioning of the hypercube was carried out automatically using the algorithm described earlier in the paper. Each pair of processors connected directly by a link communicate with each other using a commercial Ethernet (TCP/IP) protocol. Starting with different initial load configurations the load values at nodes were observed to study the rate at which the heavily loaded nodes returned to normally loaded states. Also the communication overhead incurred in this process was measured. The results of the experiments demonstrate that the hybrid algorithm performs better than the bidding and drafting algorithms. The hybrid algorithm has a faster response time as well as a lower communication overhead than either the original bidding and drafting algorithms.

In Fig. 6 the performance of the algorithms is compared in a situation where node 0 is initialized as being heavily loaded while all remaining nodes are lightly loaded. The bidding algorithm performs the worst since it is sender-initiated and node 0 is the only sender. The response time of the hybrid algorithm is slightly better than the drafting algorithm. In both the hybrid and drafting the load at node 0 falls rapidly



because there are several lightly loaded nodes that are concurrently requesting node 0 to migrate processes. This is because the drafting algorithm is receiver initiated and the hybrid algorithm also performs drafting in this situation. One might expect the drafting algorithm to have a better response time than the hybrid algorithm since in the former all seven lightly loaded nodes will be concurrently drafting node 0 while in the latter only the three nodes 1,2,3 in partition  $P_1$  will be drafting node 0. However, this is not the case because as the number of drafting processes increase so does the communication traffic and the number of draft requests that node 0 must respond to. As shown in Fig. 6 the communication overhead is greater for drafting than the hybrid algorithm.

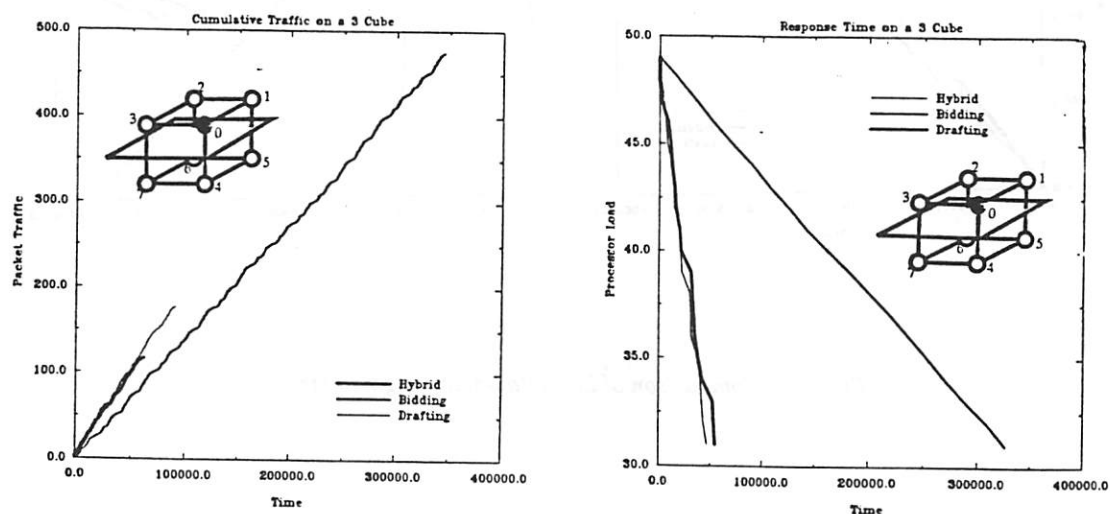


Figure 6. Comparison of Load Balancing Algorithms

In the example shown in Fig. 7 there are two heavily loaded nodes, one in each partition. Again, as expected, the performance of the bidding algorithm is the worst. This time the hybrid algorithm performs significantly better than the drafting algorithm because it only performs drafting within the partitions, which reduces communication overhead and also improves response time.

In Fig. 8 all nodes in one partition are initialized as being heavily loaded. The hybrid algorithm again performs significantly better than the bidding and drafting algorithms. The hybrid algorithm starts by performing bidding to migrate processes across the partition. Since this process is simpler and generates fewer messages than both the original bidding and drafting algorithms, the hybrid algorithm exhibits better response time. Later the hybrid algorithm switches to drafting. This process is also simple and is performed only within a partition. Thus, the hybrid algorithm performs better than the bidding and drafting algorithms.

In the example shown in Fig. 9 all nodes, except node 7 in partition  $P_2$ , are heavily loaded. In this example the performance of the bidding algorithm is better than the drafting algorithm because there are several senders and a single receiver. The hybrid algorithm gives better performance than both the bidding and drafting algorithms. Initially the hybrid algorithm performs drafting within the partition ( $P_2$ ) containing node 7 and bidding from  $P_2$  to  $P_1$ . The bidding messages are discarded by nodes in partition  $P_2$  because of the heavily loaded processors there. As a result, processes are migrated only in response to drafting by node 7. On the other hand, in the original bidding and drafting algorithms all nodes simultaneously migrate processes to node 7 creating too much traffic and a far greater number messages for node 7 to process. This slows down the response time of node 7.

In all the cases discussed in this section one can see that the hybrid algorithm performs better than the original drafting and bidding algorithms. This is both due to the simplicity of the intra-partition and

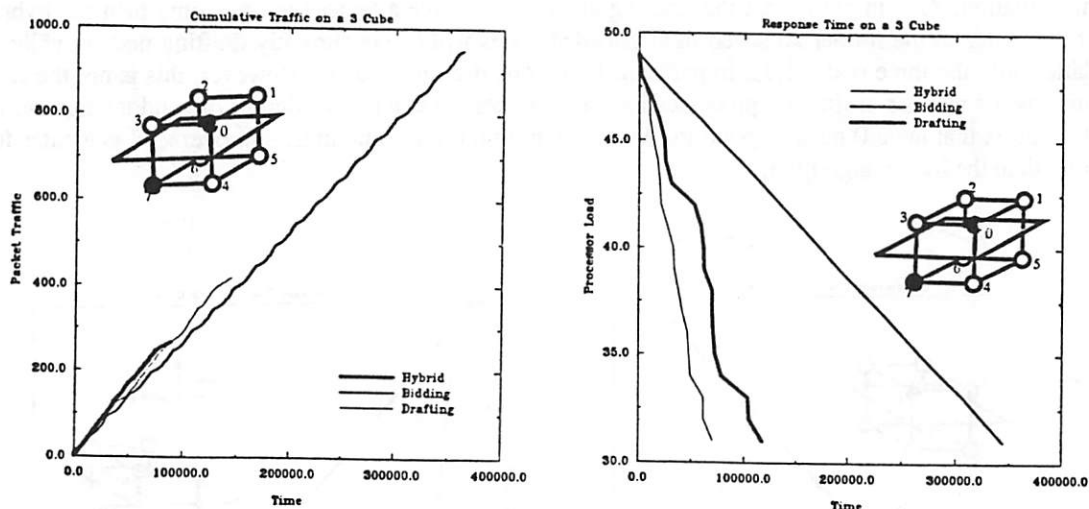


Figure 7. Comparison of Load Balancing Algorithms

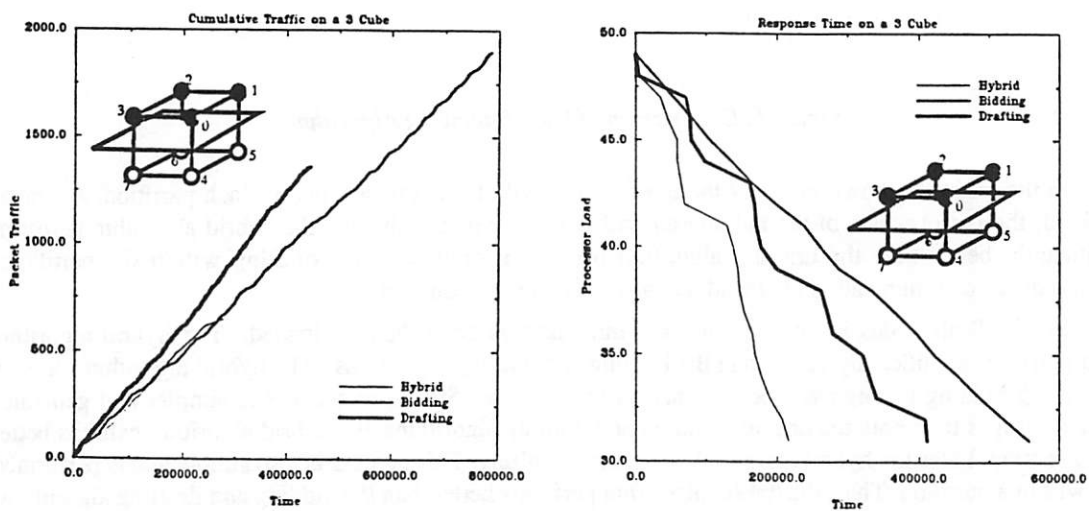


Figure 8. Comparison of Load Balancing Algorithms

inter-partition algorithms and the ability of the hybrid algorithm to switch back and forth between the inter-partition and intra-partition algorithms based upon the state of the system. Furthermore, we observe that as the number of heavily loaded processors in the system increases the improvement in the performance of the hybrid algorithm in comparison to the bidding and drafting algorithms also increases.

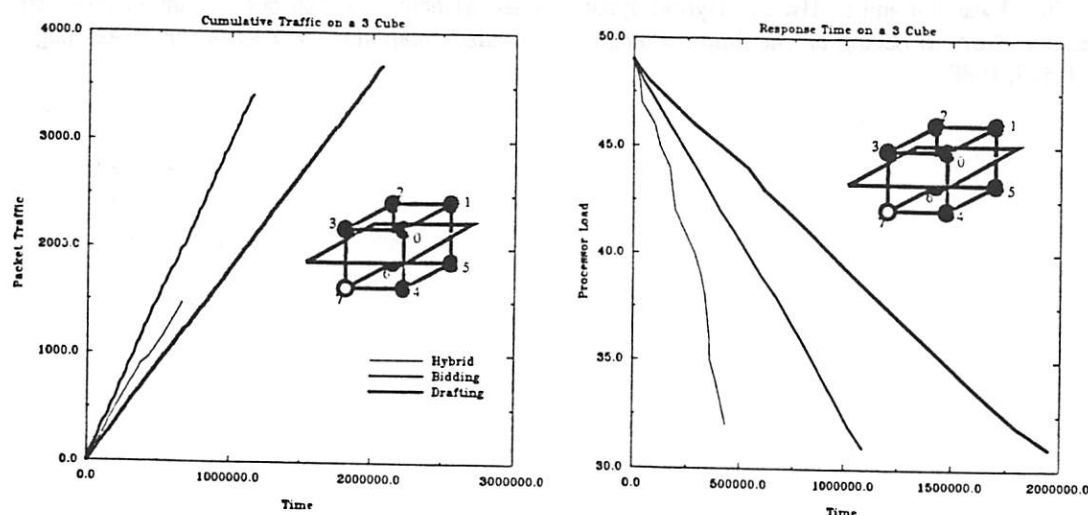


Figure 9. Comparison of Load Balancing Algorithms

## 7. Summary

In this paper a new load balancing algorithm for distributed systems was presented. A distributed system is first partitioned into disjoint partitions of desirable size. The load balancing algorithm uses a hybrid, adaptive load balancing approach consisting of intra-partition load balancing and inter-partition load balancing to ensure that process migration can be carried out even if a lightly loaded processor is separated from a heavily loaded processor by several normally loaded processors. Unlike the original bidding and drafting algorithms, this is achieved without substantially increasing the communication overhead. An intra-partition load balancing algorithm, based on drafting, is used to balance the load within the partitions. A modified form of drafting is used to obtain better response time. If this is not sufficient to achieve load balance, inter-partition load balancing is carried out. The partitioning of the system is carried out statically in a manner that tries to reduce the communication load on heavily loaded links. Simulation results demonstrate that the hybrid algorithm provides superior performance than the bidding and drafting algorithms.

## References

1. N. Hutchinson, E. Jul, H. Levy, and A. Black. Fine-grained mobility in the emerald system. *ACM Transactions on Computer Systems*, 6(1):109-133, Feb. 1988.
2. M.L. Powell and B.P. Miller. Process migration in demos/mp. In *Proceedings of the 9th Symposium on Operating System Principles*, pages 110-119. Assoc. Comput. Mach., Oct. 1983.
3. J.A. Stankovic and I.S. Sidhu. An adaptive bidding algorithm for processes, clusters and distributed groups. In *Fourth International Conference on Distributed Computing Systems, San Francisco, CA*, pages 49-59. IEEE, May 1984.
4. L.M. Ni, C-W. Xu, and T.B. Gendreau. A distributed drafting algorithm for load balancing. *IEEE Transactions on Software Engineering*, SE-11(10):1153-1161, Oct. 1985.
5. D.L. Eager, E.D. Lazowska, and J. Zahorjan. Dynamic load sharing in homogeneous distributed systems. *Technical Report*, Univ. of Washington, Oct. 1984.
6. F.C.H. Lin and R.M. Keller. Gradient model: A demand-driven load balancing scheme. *Proc. International Conf. on Distributed Computing Systems*, pages 329-336, 1986.

7. P. Gburzynski and P. Rudnicki. The LANSF protocol modeling environment. *Technical Report TR 89-19*, Univ. of Alberta, July 1989.
8. R. Chowkwanyun and K. Hwang. Hybrid dynamic load balancing for distributed-memory multiprocessors. *Proc. workshop on the future trends of distributed computing systems in the 1990s*, pages 391-399, 1988.

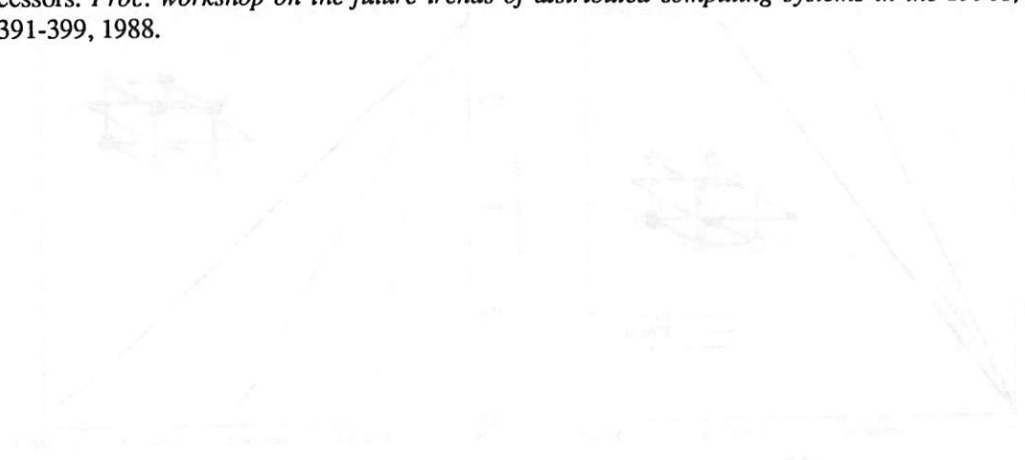


Figure 2: A diagram showing a network topology with nodes and connections, illustrating the LANSF protocol modeling environment.

Figure 2

The diagram illustrates a network topology with nodes and connections, representing a distributed system architecture. The nodes are arranged in a circular pattern with internal connections, suggesting a distributed system architecture. The diagram is labeled 'Figure 2' and is part of a technical report or paper.

Figure 2

The diagram illustrates a network topology with nodes and connections, representing a distributed system architecture. The nodes are arranged in a circular pattern with internal connections, suggesting a distributed system architecture. The diagram is labeled 'Figure 2' and is part of a technical report or paper.



## 8. Appendix A. Simulation Details.

All simulation work was done using the LANSF protocol modeling tool. LANSF is a public domain package that facilitates experimenting with different network protocols. As supplied, it is set up for broadcast bus-based systems. For our purposes we had to modify some of the sources to allow for the point-to-point communication links we require. In addition we added the code needed to perform routing of packets. The package has a fairly long learning curve, but after some experience has been gained in its use, it is very easy to experiment with different topologies. The package has a convenient user-interface with customizable displays that offer run-time monitoring of a simulation run. As an example see Figure 10 below.

```

shelltool - /bin/csh
GENERAL LINK QUEUES
PID 4717 THP 2.48 11
CPT 19 TPR 2.48 11 1
SIT314343160E3 TAT 107 11 1
EVS 9251 STF 78 211 1
MES 78 JMS 26 21111 1
UMB 20000 CAC 3 01234567
MBL 147226 BBTATWWT

TRAFFIC
Ms RcvBits VahMDelay AbsMDelay AbsPDelay AbMDelMax AbPDelMax
0 0 1 1 1 -inf -inf
1 0 1 1 1 -Inf -Inf
2 16000 3.12e+03 3.12e+03 1.86e+03 8.88e+03 4.55e+03
3 19000 5.43e+03 5.43e+03 1.31e+03 1.18e+04 1.85e+03
4 10000 2.07e+03 2.07e+03 1.28e+03 4.93e+03 1.44e+03
5 6000 2.48e+03 2.48e+03 1.25e+03 5.12e+03 1.45e+03
6 4000 4.09e+03 4.09e+03 1.28e+03 1.26e+04 1.3e+03
7 3000 1.27e+03 1.27e+03 1.24e+03 1.3e+03 1.3e+03
8 13000 5.68e+03 5.68e+03 1.8e+03 1.27e+04 4.56e+03

STRLoads
0 1 2 3 4 5 6 7
49 0 2 2 2 0 0 0
0 50 0 0 0 2 2 2
2 0 49 2 2 0 0 0
2 0 2 49 2 0 0 0
2 0 2 2 50 0 0 0
0 2 0 0 0 49 2 2
0 2 0 0 0 2 49 2
0 0 0 0 0 0 0 0

```

Figure 10. Dynamic Displays from a Simulation run



# FALCON: A Distributed Scheduler for MIMD Architectures †

Andrew S. Grimshaw  
Department of Computer Science  
University of Virginia  
Charlottesville, VA  
[grimshaw@Virginia.edu](mailto:grimshaw@Virginia.edu)

Virgilio E. Vivas Jr.  
Department of Informatics  
LAGOVEN S.A.  
Caracas, Venezuela  
[lagoven!vvivas@Sun.COM](mailto:lagoven!vvivas@Sun.COM)

## Abstract:

This paper describes FALCON (Fully Automatic Load COordinator for Networks), the scheduler for the Mentat parallel processing system. FALCON has a modular structure and is designed for systems that use a task scheduling mechanism. FALCON is distributed, stable, supports system heterogeneities, and employs a sender-initiated adaptive load sharing policy with static task assignment. FALCON is parameterizable and is implemented in Mentat, a working distributed system. We present the design and implementation of FALCON as well as a brief introduction to those features of the Mentat run-time system that influence FALCON. Performance measures under different scheduler configurations are also presented and analyzed with respect to the system parameters.

## 1. Introduction

Parallel processing has been compared to the problem of chopping up a log [Reed87]. It can be done with a large colony of termites eating the log; this corresponds to thousands of bit serial processors such as the Connection Machine. It can be done by four people with chainsaws; this corresponds to machines such as the Cray X-MP which use a small number of sophisticated processors. The middle approach is to use a small army of beavers; this corresponds to MIMD machines such as the Intel iPSC/2. Unfortunately, when taking the middle approach, it is not always sufficient to build parallel machines and to write parallel programs as collections of tasks. A further condition for success is that the tasks must be assigned to processors in such a manner as to minimize response time. Using the log analogy, if the beavers (processors) are not assigned to places on the log (tasks) wisely there may be some beavers that are idle while others are overloaded and tired. The net effect of such poor beaver (processor) management is that it takes longer to chop up the log (solve the problem).

This paper describes FALCON (Fully Automatic Load COordinator for Networks), the scheduler for the Mentat parallel processing system [Grim87, Grim90]. FALCON is a distributed, load sharing, adaptive, sender initiated, static assignment (no migration), stable scheduler designed for message passing MIMD architectures. FALCON is implemented as a collection of *instantiation managers* (*i\_m*'s). There is an *i\_m* on each processor in a Mentat system. Collectively the *i\_m*'s implement FALCON for Mentat. FALCON is parameterizable and modular, permitting experimentation with different transfer and location policies, as well as the values used in those policies. FALCON is not a paper project. It has been implemented in two different environments, a network of Sun workstations and on the Intel iPSC/2.

In this paper we first present a brief overview of the scheduling problem and Mentat. In section 2, FALCON is described. Section 3 presents the results obtained from varying FALCON policies and parameters on two different architectures, a network of Sun workstations and the Intel iPSC/2. We do not draw definitive conclusions about one policy vs. another, but rather illustrate what our experience has been with the different policies. Section 4 concludes the paper and discusses the current status of FALCON and the Mentat project.

---

† This work was partially supported by NASA contract NAG-1-1181 and by LAGOVEN S.A., Caracas, Venezuela.

## 1.1. Background on Scheduling

The *scheduling function* is both a mechanism and a policy that manages the access and utilization of a resource by its various consumers. There are two important properties which must be considered in implementing and evaluating a scheduling system. First, there is the satisfaction of the consumers with how well the scheduler manages the resource in question (performance). Second, there is the satisfaction of the consumers in terms of how difficult or costly it is to access the management resource itself (efficiency). This general description of the scheduling problem can be easily applied to the management of processors in distributed systems.

The purpose of task scheduling is to take a single job composed of multiple tasks and assign them to different processors in order to maximize performance. Tasks are considered to be consumers which use the processors of the system. The distributed scheduler manages the processors. A task should be able to quickly and efficiently access its assigned resources, and a task should not be hindered by scheduling overhead.

There is a rich literature in the area of scheduling in distributed systems [Boel89, Casa88, Diks89, Eage86a-b, Hac89, Hsu86, Lo84, Stan85, Tanc86, Tant85, Trip88]. The work can usually be classified as theoretic or heuristic. Theoretic approaches tend to require *a priori* information about the tasks to be scheduled such as the resource requirements (CPU, memory) of the task, and the IPC costs between every pair of tasks. Even with this *a priori* information the problem of finding an optimal assignment of tasks to processors is NP-Hard [Hsu86].

*Load sharing* schedulers are heuristic schedulers that attempt to improve the performance of a distributed system by using the processing power of the entire system to alleviate periods of high congestion at individual nodes. This is done by transferring some of the workload of a congested node to other nodes for processing. Load sharing policies do not attempt to find optimal schedules, instead they attempt to find "good" schedules. The "goodness" of the schedule usually depends on the amount of effort that goes into finding it. Thus, there is a trade-off between the quality of the schedule and the effort to find it. Some [Eage86b] have argued that in most cases it is best to make only a minimal effort at finding the best schedule, that simple algorithms yield good results. We have taken this approach in FALCON.

## 1.2. Mentat Overview

Mentat is an object-oriented, parallel computation system designed to provide large amounts of easy to use parallelism for distributed systems. Mentat combines a medium-grain, data-driven computation model with the object-oriented programming paradigm and provides automatic detection and management of data dependencies. The data-driven computation model [Liu86a-b] supports high degrees of parallelism and a simple decentralized control, while the use of the object-oriented paradigm permits the hiding of much of the parallel environment from the programmer. Because Mentat uses a data-driven computation model, it is particularly well-suited for message passing, non-shared memory architectures.

There are two primary aspects of Mentat: the Mentat Programming Language (MPL) [Grim88] and the Mentat run-time system [Grim87, Grim90]. The Mentat Programming Language, an extension of C++ [Stro86], simplifies writing parallel programs by extending the encapsulation provided by objects to the encapsulation of parallelism. Users of Mentat objects are unaware of whether member functions are carried out sequentially or in parallel. In addition, member function invocation is asynchronous (non-blocking): the caller does not wait for the result. It is the responsibility of the compiler, in conjunction with the run-time system, to manage all aspects of communication, synchronization, and scheduling. The underlying assumption is that the programmer can make better granularity and partitioning decisions, while the compiler and run-time system can correctly manage communication and synchronization. By splitting the responsibility between the compiler and the programmer we exploit the strengths of each, and avoid their weaknesses.



The run-time system supports parallel object-oriented computing on top of a data-driven, message-passing model. It supports more than just method invocation by remote procedure call [Nels81]. Instead the run-time system supports a graph-based, data-driven computation model in which the invoker of an object member function need not wait for the result of the computation, or for that matter, ever receive a copy of the result. The run-time system constructs program graphs, and allows selective message reception. Furthermore, the run-time system is portable across a wide variety of MIMD architectures and runs on top of the existing host operating system. The underlying operating system must provide process support and some form of inter-process communication.

The Mentat run-time system architecture consists of a set of processors communicating through some interconnection network. Each processor has a complete copy of the run-time system. The run-time system can be split into two parts: the library routines that are linked with each Mentat object (user application), and the *instantiation manager* (*i\_m*). Each *instantiation manager* is a separate process.

The *i\_m*'s implement FALCON and perform the object management function. Each *i\_m* executes a server loop, accepting service requests. The service requests invoke one of several functions. The function of interest here is that of an instantiation request. An instantiation request contains the name of the class to instantiate and, optionally, some *location hints*. The instantiation request invokes the scheduling function. (Henceforth we will use the term *task* to refer to an instantiation request.) Location hints are used by the *i\_m* in making the scheduling decision.

FALCON deals exclusively with global scheduling, i.e., deciding *where* to execute a process. The job of local scheduling is left to the underlying host operating system. The Mentat computation model requires a *task scheduling* mechanism. Each object, or actor, is treated as an independent task of a program by the task scheduling mechanism.

Mentat has been implemented on three architectures that span the MIMD spectrum, a network of Sun workstations (loosely coupled), the Intel iPSC/2 (tightly coupled), and the BBN Butterfly (shared memory). Mentat programs are source compatible between supported architectures. Even on an eight processor network of Sun workstations, speed-ups in excess of four, in comparison to optimized sequential C code, are consistently seen for Gaussian elimination using partial pivoting.

## 2. FALCON

FALCON is based upon the work of Eager, Lazowska and Zahorjan [Eage86a, Eage86b and Eage88], who developed a model for *sender-initiated adaptive load sharing* for homogeneous distributed systems. Their model uses system state information to describe the way in which the load in the system is distributed among its components.

Mentat was designed to work on system architectures and topologies in which processors do not necessarily have equal performance characteristics. Further, the Mentat programming language (MPL) provides information on process characteristics that can be used by FALCON to improve its decisions. These design features make the pure Eager, Lazowska and Zahorjan model insufficient. We built a new model, based on theirs, to accommodate these differences.

FALCON falls into the classification schemes of [Casa88] and [Hac89] as having the following characteristics:

1. **Distributed:** The scheduling decision is distributed. Each decision is reached independently of all others.
2. **Load Sharing:** FALCON transparently distributes the workload by transferring new tasks from nodes that are heavily loaded to nodes that are lightly loaded.
3. **Adaptive:** FALCON employs information on the current system state in making transfer decisions. Hence, it reacts to changes in the system state.
4. **Sender-Initiated:** Congested nodes search for lightly loaded nodes.

5. **Static Assignment:** Each task remains on the node to which it is assigned until the execution of the task and communication with other tasks is completed. In other words, there is no migration or reassignment after execution of a task is started.
6. **Stable:** A task can only be transferred a fixed number of times between the nodes of the system. Transfers occur before the beginning of task execution while searching for a suitable node to handle the task. Hence, processor thrashing is avoided.

Each *i\_m* consists of three major modules: 1) detection of *location hints*, 2) determining whether to process a task locally or remotely (*transfer policy*), and 3) determining to which processor a task selected for transfer should be sent (*location policy*).

The modular structure of the *i\_m*'s facilitate the addition of, and experimentation with, transfer and location policies. Thus, for instance, the *i\_m*'s could be used as a *testbed* for evaluating load sharing policies [Diks89]. We have used this capability to experiment with a variety of transfer and location policies.

Collectively the *instantiation managers* implement FALCON and provide it with the following features:

1. The scheduling decision is divided into two components: the *transfer policy* that determines whether to process a task locally or remotely, and the *location policy* that determines where a task selected for transfer should be sent.
2. The transfer policy can use the run queue length of the processors and other information about the system state such as available memory and percentage of CPU utilization.
3. FALCON uses a threshold transfer policy and allows a choice between different location policy algorithms with a static transfer limit. Currently there are three selectable location policies, *random*, *round-robin* and *best-most-recently*. A static transfer limit makes our model stable and prevents processor thrashing.
4. FALCON may be configured for different architectures. The configuration is based on the state information that the host operating system supplies to the scheduler. Some systems offer easy and quick access to their state information (e.g., SunOS [Sun88]), whereas others (e.g., Intel's NX/2 [Inte88]) do not provide an easy way to collect this information.
5. FALCON supports different system topologies and processor power heterogeneities via the creation of *logical sub-networks* of homogeneous processors. These logical sub-networks also model separate physical sub-networks.
6. FALCON may be configured for different applications. When the behavior of an application is previously known, the scheduler can be tuned to better accommodate the application. This is done by considering the number of pieces in which the application is divided, their granularity, and their resource utilization (memory and cpu).
7. FALCON uses programmer specified *location hints* to improve scheduling decisions.

## 2.1. Function Modules

The *i\_m* consists of three major components (see Figure 1). The components 1) detect and act upon *location hints*, 2) determine whether to process a task locally or remotely (*transfer policy*), and 3) determine to which processor a task selected for transfer should be sent (*location policy*). Each of these three functions is described below.

### 2.1.1. Detection of Location Hints

The Mentat Programming Language allows the programmer of a Mentat application to specify *location hints* [Grim89]. By doing this, the programmer can give FALCON information that is used in making scheduling decisions. Detection of location hints is the first function performed by the *i\_m*. In the absence of location hints the *i\_m* forwards the request to module 2.

There are currently three types of location hints which can be specified using MPL. They are CO-LOCATE, DISJOINT and HIGH-COMPUTATION-RATIO. Depending on the location hint specified by the programmer, additional information is required by the *i\_m* in order to

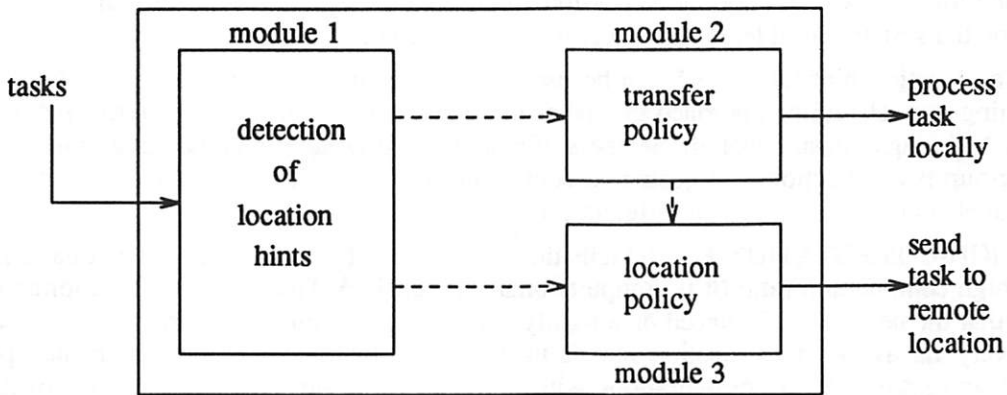


Figure 1. Instantiation Manager components and their interaction.

make its decisions (see Figure 2). CO-LOCATE tells the  $i\_m$  to locate the object being instantiated close enough to another object so that communication between the two is inexpensive. Therefore, CO-LOCATE needs to supply the name of that other object as input to the  $i\_m$ . The object name contains its location information. With this information in hand, the  $i\_m$  places the new object on the node where the other object is instantiated.

Lo[84] describes *clustering*, in which closely related groups of processes are placed on the same processor to reduce interprocess communication costs. CO-LOCATE can be used to indicate the presence of a cluster. A programmer may decide to use CO-LOCATE on objects that will exchange a large amount of data or on objects that will exchange data frequently.

Location Hint	Information Required	Action
<b>CO-LOCATE</b>	Name of an instantiated object.	The scheduler places the object on the processor where the named object is instantiated.
<b>DISJOINT</b>	List of object names.	The scheduler avoids placing the object on a processor where any object from the list is instantiated.
<b>HIGH-COMPUTATION-RATIO</b>	None.	The scheduler places the object on a lightly loaded, powerful or distant processor.

Figure 2. Location hints.

DISJOINT tells the *i\_m* that the object to be instantiated should *not* be instantiated on the same processor as any object in a given list of objects. The *i\_m* uses the list of object names to select an appropriate node for instantiation. Any node which is not in the list of object names is a candidate for selection. It may not be possible to instantiate disjointly, if, for instance, every processor on the system has at least one object from the list on it.

The location hint DISJOINT can be used by programmers to achieve co-scheduling. Co-scheduling is a scheduling approach that takes interprocess communication patterns into account while scheduling to ensure that all members of a *distributed group* run at the same time. A distributed group is a collection of objects that communicate with each other but would benefit from the parallelism of being placed on different processors.

HIGH-COMPUTATION-RATIO tells the *i\_m* that the object to be instantiated has a particularly high computation ratio (it is computationally expensive). The *i\_m* uses this information to ensure that the new object is placed on a lightly loaded or powerful processor, even if the processor is very far away. This location hint is useful on configurations of heterogeneous (performance) processors and/or configurations with more than one physical sub-network. In heterogeneous systems, the processors differ in power capabilities and this location hint tends to find the most powerful processor in the system. In configurations of more than one physical sub-network, a processor in a different sub-network is considered as a 'far away' processor. The *i\_m* sends high-computation-ratio tasks to different sub-networks. In homogeneous systems, FALCON searches for a lightly loaded processor.

### 2.1.2. Transfer Policy

*Transfer policy* corresponds to the second module in Figure 2. This module is executed for tasks that do not have a location hint, and for tasks in which the location hint includes the current node as a candidate for execution. Transfer policy in a load sharing model determines whether to process a task locally or remotely. The policy that we have selected is a **threshold** policy: a distributed, adaptive policy in which each node of the system uses only local state information to make its decisions. *No exchange of state information among the nodes is required in deciding whether to transfer a task.* A task originating at a node is accepted for processing if the local state of the system is below some threshold *T*. Otherwise, an attempt is made to transfer that task invocation request to another node. Note that only newly received tasks are eligible for transfer.

Selection of a value for *T* is based on the local state information that the host operating system supplies to the *i\_m*. Also, if the behavior of an application is known in advance, the value of *T* can be chosen to improve the application performance. The threshold *T*, as well as the state information to which it corresponds, is configurable.

An issue in the implementation of a transfer policy is how the destination node should treat an arriving transferred task. One answer is that it should treat it just as a task originating at the node: if the local state of the system is below threshold, the task is accepted for processing. Otherwise, it is transferred to some other node selected by the location policy algorithm. This can cause instability; the system may eventually enter a state in which the nodes are devoting all of their time to transferring tasks and none of their time to processing them.

Instability is overcome by the use of an appropriate control policy. The simple control policy that we use is to impose a static limit on the number of times that a task can be transferred. The destination node of the last allowed transfer must process the task regardless of its state. When the transfer limit is reached the *i\_m* must accept the task. The transfer limit depends on the number of available nodes and may be configured.

### 2.1.3. Location Policy

The location policy is invoked if the transfer policy does not accept the object for local instantiation or if a location hint that excludes the residing node as a candidate for execution of



the object has been specified. The three location policy algorithms that we have implemented are *random*, *round-robin* and *best-most-recently*.

#### 2.1.3.1. Random Algorithm

The simplest location policy is one that uses no information at all. With the *random* policy a destination node is selected at random and the object is transferred to that node. No exchange of state information among the nodes is required in deciding where to transfer an object. FALCON relies on the *drand48* library function to obtain random numbers (with an appropriate seed). The generated number is multiplied by the number of nodes in the system and the result is rounded. The resulting number represents a node in the system. In addition, the random location policy avoids certain destination nodes. Since the object was not accepted locally the current node is not a candidate. Further, if an object was received from another node, the object should not be sent back because it was previously rejected.

#### 2.1.3.2. Round-Robin Algorithm

Another simple location policy that uses no state information is the *round-robin* policy. Under the *round-robin* policy, destination nodes are selected in a statically determined sequence. No exchange of state information among the nodes is required to decide where to transfer an object. Nodes are arranged in a circular queue. Each *i<sub>m</sub>* remembers the last node it chose as a destination node. When a task enters the location policy, its destination becomes the next node of the circular queue. Then, the last node is updated with the new destination.

#### 2.1.3.3. Best-Most-Recently Algorithm

This location policy acquires and uses a small amount of system state information and attempts to make the 'best' choice given this information. When a task enters the location policy, its immediate destination is determined randomly. Suppose a task does not find a suitable node for execution before its transfer limit expires. Then, the best-most-recently algorithm uses the last transfer to send the task back to the most recently visited node that had the 'best' system state. The node with the 'best' system state is the closest to the threshold *T*. FALCON knows the 'best' node because each time a task is transferred, the local state information is sent with the task. If the system state of the local node is better than or equal to the current 'best' node, the local node becomes the 'best'. Otherwise, the 'best' node is left as it was received.

The advantage of this location policy is that it uses information about the state of the system. FALCON balances the load among the nodes of the system because it always selects the least busy sampled node to handle the requests.

One disadvantage of this location policy is that race conditions can occur. If there are two tasks searching for a node, at the end of their transfer limits the 'best' node could be the same for both. Hence, the independent schedulers will send both tasks to the same node. This could cause an overload on that specific node. Furthermore, the system state at the 'best' node could have changed by the time the object gets back because of external (to Mentat) processes. To overcome this disadvantage we allow an additional verification when the object gets back to the 'best' node. If the new state of the 'best' node is better than or equal to the old state, the object is accepted. Otherwise, the object is returned to the node at which the transfer limit expired.

### 2.2. Logical Sub-Networks

Another feature supported by FALCON is the creation of *logical sub-networks*. Logical sub-networks are motivated by the difficulty of handling heterogeneous environments. Our scheduling policies do not consider either the processing power of the nodes or the communication cost when making scheduling decisions. They assume that all the nodes in the system are homogeneous and the communication cost is the same to any node of the system. In order to

maintain these assumptions, logical sub-networks group homogeneous nodes which are interconnected in the same physical network.

In Figure 3, a physical network of two physical sub-networks is split into logical sub-networks. In the physical sub-network on the left side of the figure there are two logical sub-networks: one is grouping Sun 3/75s and the other Sun 3/60s. In the other side of the figure, there are three logical sub-networks; one of Sun 3/75s, another of Sun 3/60s, and another of Sun 3/50s. The grouping of homogeneous nodes establishes an ordering among the nodes of the system. The first logical sub-network consists of the most powerful processors. The next sub-network consists of the second most powerful processors. The third sub-network, consists of the third most powerful processors or an adjacent physical sub-network. Therefore, the entire system would be formed by sub-networks ordered by processing power and/or network distance.

An important consequence of structuring an entire system into logical sub-networks is that applications can be separated from each other by placing them in different sub-networks and restricting inter-sub-network scheduling. Further, for management purposes, different sub-networks can be used as testing and/or production environments. We have found this to be very useful.

FALCON uses the logical sub-networks in two ways. First, FALCON may use an additional transfer limit that indicates how many sub-networks are to be considered for scheduling tasks. If an object consumes its internal transfer limit in a sub-network, it can be sent to another sub-network and remain there until it is instantiated. Once an object leaves a sub-network, it never returns. This is necessary for stability.

Second, logical sub-networks may be used to support location hints. HIGH-COMPUTATION-RATIO requires the knowledge of processing power and/or distance between nodes. Logical sub-networks indirectly give this knowledge to FALCON. If the location decision is 'instantiate far away', the object is sent to a different sub-network. If the location decision is 'powerful node', the object is sent to the sub-network with the most powerful nodes.

In this section we have examined how the  $i_m$  breaks the scheduling decision into three sub-tasks, interpretation of location hints, implementation of the transfer policy, and implementation of the location policy. The different policies and environmental combinations, shown in

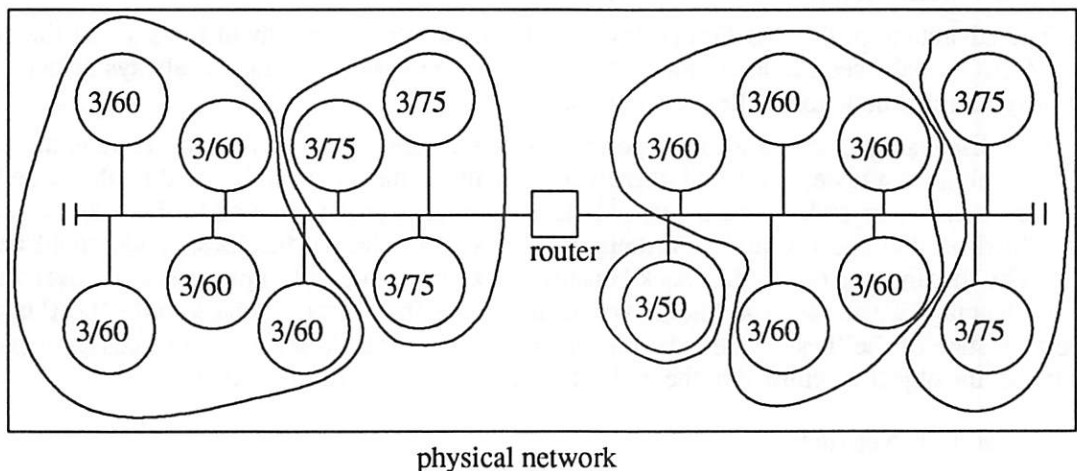


Figure 3. Logical sub-networks.

		SUN network	Hypercube
<i>Location hints</i>	CO-LOCATE	The scheduler places the object on the processor where the named object is instantiated.	
	DISJOINT	The scheduler avoids placing the object on a processor where any object from the list is instantiated.	
	H-C-R	The scheduler places the object on a lightly loaded, powerful or distant processor, depending on number of logical sub-networks.	The scheduler places the object on a lightly loaded processor.
<i>Transfer policy (Threshold)</i>		The scheduler might use running queue length, free memory or % of idle cpu as its local system state.	The scheduler only uses Mentat objects' queue length as its local system state.
<i>Location policy</i>	Random	If transfer limit expires, the object could be sent to another sub-network.	If transfer limit expires, the object remains on the local sub-network.
	Round-Robin		
	B-M-R		
<i>Logical sub-networks</i>		More than one are allowed; nodes can be heterogeneous.	Only one is allowed; nodes are homogeneous.

Figure 4. Model application on environments.

Figure 4, exploit available system state information, the valid thresholds, and the use of logical sub-networks. Future implementations are expected to vary in the same manner, but they will maintain and support the same structure.

### 3. Analysis of Results

The scheduler described in the previous section is not a paper project. We have implemented FALCON in two different environments, a network of Sun workstations and on the Intel iPSC/2. In this section we present results obtained by varying the scheduling policies and parameters. We recognize that the results depend heavily on the applications chosen and, in the case of the network of Suns, on other traffic and jobs. Our goal is not to prove or disprove that one policy is better than another under all circumstances; rather it is to illustrate that FALCON works and to give some insight into the sensitivity of the policies to their parameters.

In the following figures, data for one application, Gaussian elimination of a 300X300 matrix using partial pivoting, is shown. The data are representative of what we have experienced. Speedup is relative to an equivalent C program, not relative to the Mentat implementation running on one processor. We have chosen to compare the performance against sequential C because users of parallel architectures are interested in raw performance, i.e., the benefit gained by using parallel code. Gaussian elimination is used because it is the *de facto* parallel processing benchmark. The maximum speedup for Gauss is seen when the number of pieces, i.e., tasks of an application, equals the number of nodes in the system. When a matrix is split into  $n$  pieces, our Mentat implementation instantiates  $n+1$  objects. The extra object splits the matrix, transmits its

<sup>1</sup>The complete data on several applications, under a wide variety of policies and parameters, are available in [Vivas90]. Additional data on performance and evaluation is available in [Cheek90].

components, and assembles the solution. Therefore, the peak speedup (5.44) is achieved when the matrix is split in seven pieces. FALCON should keep one piece of the matrix on each node until the number of pieces becomes greater than the number of nodes. The drop on the curve at eight pieces is caused because at this time there are nine objects of software to be located on eight nodes. Thus two objects share one processor. From this point on, FALCON balances the load among the nodes of the system, and the rest of the curve is stable.

The data were collected in two different environments: a network of SUN Workstations, and an Intel Hypercube. The network of SUNs consists of eight Sun 3/60's serviced by a Sun 3/280 file server running NFS. The interconnection network is a thin Ethernet. All the workstations have eight megabytes of memory. The run-times used to compute the speedups for the Suns were obtained late at night, when the network load was relatively light. We chose to work with eight nodes in both environments.

The Intel iPSC/2 is configured with eight 80386 nodes. Each node has four megabytes of physical memory and an 80387 math co-processor. The nodes were **not** equipped with either the VX vector processor or with the SX scalar processor. The NX/2 operating system provided with the iPSC/2 does not support virtual memory. The lack of virtual memory, coupled with the amount of memory consumed by the OS, limited the problem sizes we could run on the iPSC/2. Contention with other users was not an issue: once Mentat acquired a cube, the cube could only be used by Mentat objects.

### 3.1. Choice of Threshold

The transfer policy that we have selected is a *threshold* policy: a distributed, adaptive policy in which each node of the system uses only local state information to make its decisions. An object originating at a node is accepted for processing if the local state of the system is below some threshold  $T$ . Choosing the appropriate value  $T$  depends on the environment, the running application and the transfer cost of tasks in the system. It depends on the environment because of the local state information that the host operating system supplies to FALCON. It depends on the application application graph structure because different graphs have different requirements. Finally, it depends on the transfer cost because a very small  $T$  results in many transfers.

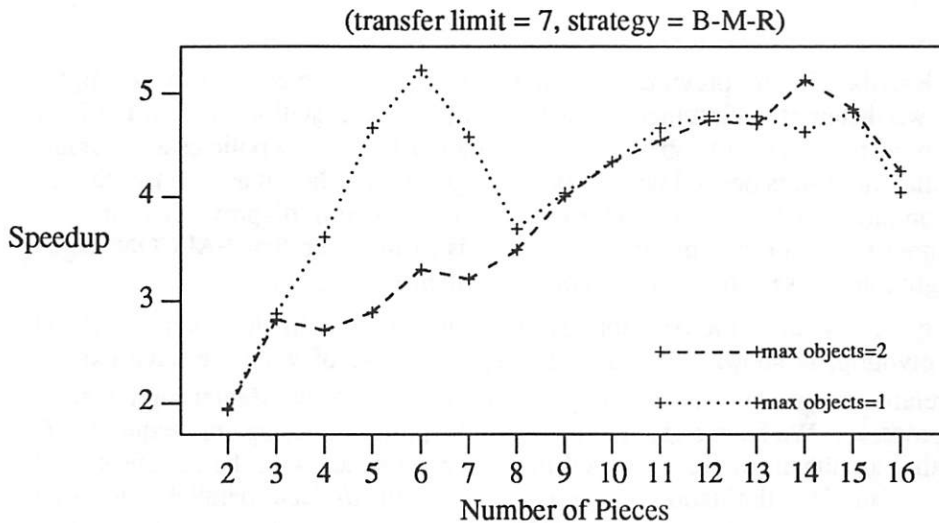


Figure 5. 8-node Hypercube: Choice of threshold value.

Figure 5 shows the relation between the threshold value, in this case the number of Mentat objects, and the speedup achieved by the application. It can be seen that a threshold value of 1



gives the better overall performance. This is because the cost to find a lightly loaded processor is small relative to the amount of time spent performing computation.

Other parameters can be used in the transfer policy if they are available. On the Sun workstations we have used CPU % idle, average length of the run queue, and the amount of free memory. Figure 6 illustrates the results. Free memory was consistently the worst performer. CPU % idle and run queue length are almost the same. This makes sense because CPU % idle is computed from the average run queue length over an interval.

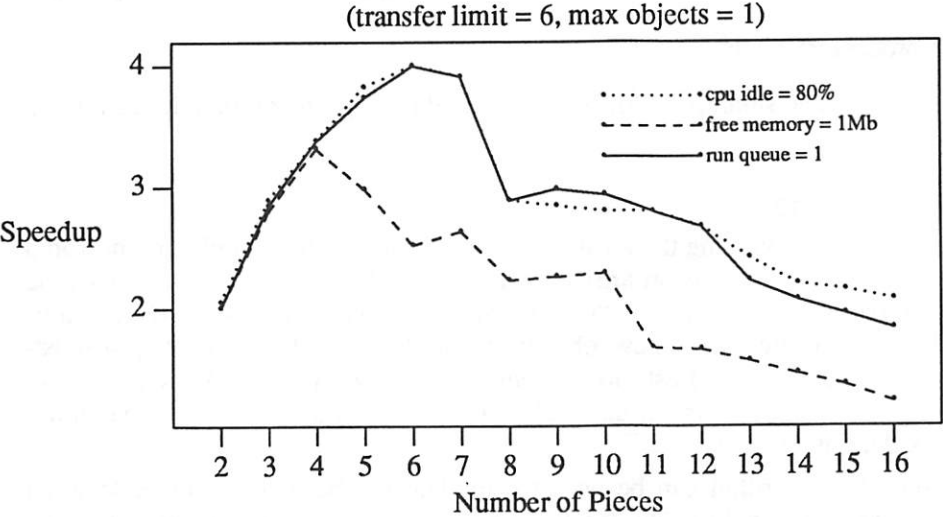


Figure 6. Network of Sun Workstations: Different threshold parameters.

### 3.2. Choice of Transfer Limit

The transfer limit is used to prevent processor thrashing and to make our scheduling mechanism stable. The value of this parameter directly affects the transfer cost. A large value for the transfer limit increases the transfer cost. However, a large value helps FALCON in finding the best place to schedule a task.

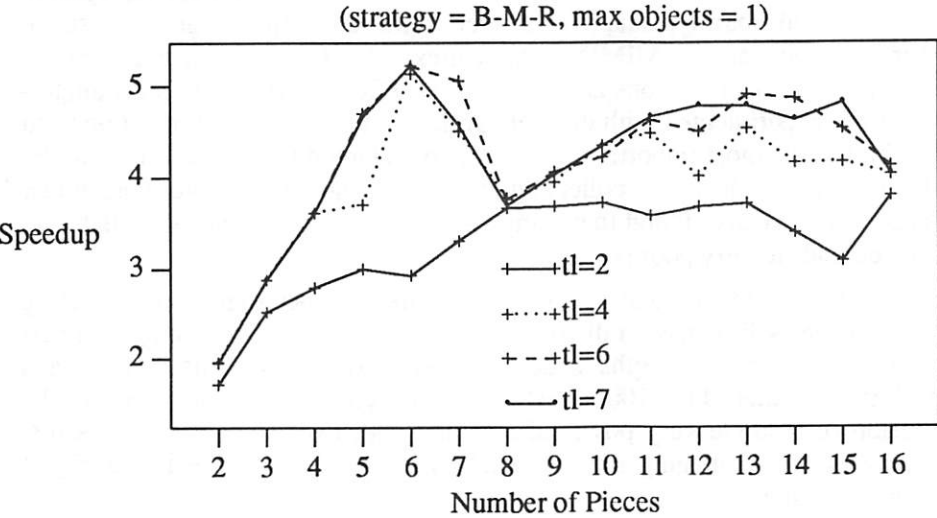


Figure 7. 8-node Hypercube: Choice of transfer limit.

Figure 7 shows how the transfer limit affects application performance. The largest possible value for the transfer limit is one less than the number of nodes. It can be seen from the figure that increasing the limit beyond six only marginally improves performance. In general there is a relationship between the optimum limit, the number of busy processors, and the number of processors. This makes sense. For example, assume that  $b\%$  of processors are busy (above threshold). Then the probability of finding an idle processor with  $p$  probes is:

$$P(\text{finding an idle processor}) = 1 - (\text{probability of choosing a busy processor on each of } p \text{ probes})$$

$$P(\text{finding an idle processor}) = 1 - b^k.$$

Thus, for low loads (small  $b$ ) and small  $p$ , or for high loads and large  $p$ , the probability of finding an idle node is high.

### 3.3. Choice of Location Strategy

Figure 8 shows the result of varying the location policy. At low transfer limits, round-robin is best. The superiority of round-robin is an artifact of the application; in this example a single node generates all instantiation requests, hence round-robin evenly distributes them. Other applications that dynamically generate many new objects at run-time do better with best-most-recently, particularly under heavy load. Best-most-recently does better under heavy load (e.g., in Figure 8 when the number of pieces is greater than 8) because it keeps track of the best load it has seen and sends the task there in the end.

The best-most-recently algorithm can be used to simulate the behavior of other location strategies. By setting a small threshold value, the tasks will be forced to go from node to node until the transfer limit is reached. The transfer limit can be considered a probe limit in this case because the limit indicates the number of nodes that are consulted for state information. The Shortest Queue Length policy can be simulated by setting the maximum number of Mentat objects to 0. This forces the local  $i\_m$  schedulers to reject the task until the transfer limit is reached. At this point the node with the shortest queue length is known and the task is sent to it.

## 4. Summary and Future Work

This paper has described FALCON, the scheduler for the Mentat parallel processing system. FALCON is a distributed, load sharing, adaptive, sender initiated, static assignment, stable scheduler designed for message passing MIMD architectures. FALCON is currently implemented on both a network of Sun workstations and on the Intel iPSC/2. FALCON is parameterizable and modular. We have experimented with different policy and parameter combinations and found that the transfer limit is the most important parameter for achieving a balanced load under moderate to heavy loads. This implies that collecting more information is appropriate under moderate to heavy loads. We have also found that using the amount of free memory available as a transfer policy parameter leads to very poor performance.

Currently we are modifying Mentat and FALCON to work in a heterogeneous operating system environment. FALCON will manage a distributed computing system consisting of a network of Sun workstations of various strengths under SunOS, Next workstations under Mach [Acce86], and a BBN Butterfly under Mach-1000 [BBN88]. When complete Mentat users will be presented with the illusion of a single very powerful machine with over sixty five processors. FALCON will be responsible for managing the computation resources, and scheduling object computations throughout the system.

For more information on Mentat, or a copy of Mentat, write the author at [grimshaw@Virginia.edu](mailto:grimshaw@Virginia.edu), or call (804)-982-2204. Mentat is available for the systems described above. In addition, Mentat can be easily ported to most other Unix systems that support UDP datagrams. A C++ compiler is necessary to port Mentat or to develop Mentat applications.

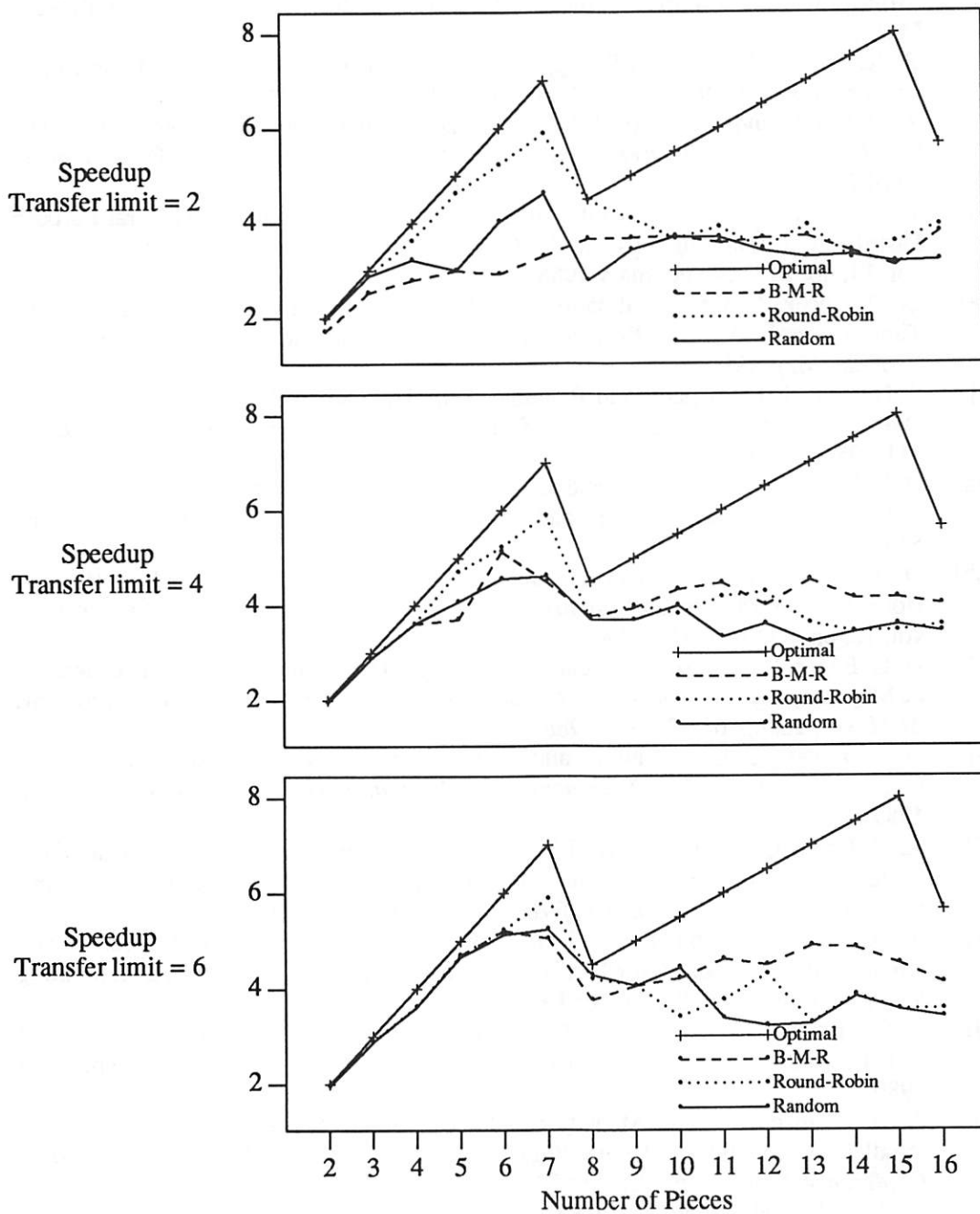


Figure 8. 8-node Hypercube: Location algorithms varying transfer limit.

## References

- [Acce86] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A New Kernel Foundation for Unix Development", *Summer Usenix Conference Proceedings*, pp. 93-112, 1986.

- [Baum89] K. M. Baumgartner and B. W. Wah, "GAMMON: A Load Balancing Strategy for Local Computer Systems with Multiaccess Networks", *IEEE Transactions on Computers*, vol. 38, pp. 1098-1109, August 1989.
- [BBN88] BBN Advanced Computers Inc., "Mach-1000 Reference Manual", Cambridge, Mass., 1988.
- [Boel89] R. K. Boel and J. H. Van Schuppen, "Distributed Routing for Load Balancing", *Proceedings of the IEEE*, vol. 77, pp. 210-221, January 1989.
- [Brya81] R. M. Bryant and R. A. Finkel, "A Stable Distributed Scheduling Algorithm", *The 2nd International Conference on Distributed Computing Systems*, Paris, France, April 1981.
- [Casa88] T. L. Casavant and J. G. Kuhl, "A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems", *IEEE Transactions on Software Engineering*, vol. 14, pp. 141-154, February 1988.
- [Check90] G. P. Check, "System Validation and Performance Analysis of the Mentat Run-Time System", *Masters Project, Department of Computer Science*, University of Virginia, May, 1990.
- [Diks89] P. Dikshit, S. K. Tripathi and P. Jalote, "SAHAYOG: A Test Bed for Evaluating Dynamic Load-Sharing Policies", *Software-Practice and Experience*, vol. 19, pp. 411-435, May 1989.
- [Eage86a] D. L. Eager, E. D. Lazowska and J. Zahorjan, "A Comparison of Receiver-Initiated and Sender-Initiated Adaptive Load Sharing", *Performance Evaluation*, vol. 6, pp. 53-68, March 1986.
- [Eage86b] D. L. Eager, E. D. Lazowska and J. Zahorjan, "Adaptive Load Sharing in Homogeneous Distributed Systems", *IEEE Transactions on Software Engineering*, vol. 12, pp. 662-675, May 1986.
- [Eage88] D. L. Eager, E. D. Lazowska and J. Zahorjan, "The Limited Performance Benefits of Migrating Active Processes for Load Sharing", *Performance Evaluation Review, ACM*, vol. 16, pp. 63-72, May 1988.
- [Eage89] D. L. Eager, E. D. Lazowska and J. Zahorjan, "Speedup Versus Efficiency in Parallel Systems", *IEEE Transactions on Computers*, vol. 38, pp. 408-423, March 1989.
- [Grim87] A. S. Grimshaw and J. S. W. Liu, "MENTAT: An Object-Oriented Data-Flow System", *Proceedings of the 1987 Object-Oriented Programming Systems, Languages and Applications Conference, ACM*, pp. 35-47, October 1987.
- [Grim88] A. S. Grimshaw and J. W. Liu, "The Mentat Programming Language and Architecture", *Proceedings Workshop on Future Trend of Distributed Computing Systems*, Hong Kong, September 1988.
- [Grim89] A. S. Grimshaw and E. Loyot, "The Mentat Programming Language: Users Manual and Tutorial", *Computer Science TR-89-06*, University of Virginia, September 1989.
- [Grim90] A. S. Grimshaw, "The Mentat Run-Time System: Support for Medium Grain Parallel Computation", *Proceedings of the Fifth Distributed Memory Computing Conference*, Charleston, SC., April 1990.
- [Hac89] A. Hac, "Load Balancing in Distributed Systems: A Summary", *Performance Evaluation Review, ACM*, vol. 16, pp. 17-25, February 1989.
- [Hsu86] C. H. Hsu and J. W. Liu, "Dynamic Load Balancing in Distributed Systems", *The 6th International Conference on Distributed Computing Systems*, Cambridge, MA, May 1986.
- [Inte88] Intel Corporation, "iPSC/2 USER'S GUIDE", Intel Scientific Computers, Beaverton, OR, March 1988.
- [Lela86] W. Leland and T. Ott, "Load-Balancing Heuristics and Process Behavior", *Proceedings of Performance '86 and ACM SIGMETRICS 1986*, pp. 54-69, May



- 1986.
- [Liu86a] J. W. Liu and A. S. Grimshaw, "A Distributed System Architecture Based on Macro Data Flow Model", *Proceedings Workshop on Future Directions in Architecture and Software*, South Carolina, May 1986.
  - [Liu86b] J. W. Liu and A. S. Grimshaw, "An Object-Oriented Macro Data Flow Architecture", *Proceedings of the 1986 National Communications Forum*, September 1986.
  - [Lo84] V. M. Lo, "Heuristic Algorithms for Task Assignment in Distributed Systems", *The 4th International Conference on Distributed Computing Systems*, San Francisco, California, May 1984.
  - [Nels81] B. J. Nelson, "Remote Procedure Call", Xerox Corporation Technical Report CSL-81-9, May 1981.
  - [Reed87] D. Reed and R. Fujimoto, *Multicomputer Networks: Message-Based Parallel Processing*, The MIT Press, Cambridge, Mass., 1987.
  - [Shen88] S. Shen, "Cooperative Distributed Dynamic Load Balancing", *Acta Informatica*, vol. 25, pp. 663-676, June 1988.
  - [Shin89] K. G. Shin and Y.-C. Chang, "Load Sharing in Distributed Real-Time Systems with State-Change Broadcasts", *IEEE Transactions on Computers*, vol. 38, pp. 1124-1142, August 1989.
  - [Shu89] W. Shu and L. V. Kale, "Dynamic Scheduling of Medium-Grained Processes on Multicomputers", *Technical Report*, report no. UIUCDCS-R-89-1528, University of Illinois at Urbana-Champaign, July 1989.
  - [Stan84] J. A. Stankovic and I. S. Sidhu, "An Adaptive Bidding Algorithm for Processes, Cluster and Distributed Groups", *The 4th International Conference on Distributed Computing Systems*, San Francisco, California, May 1984.
  - [Stan85] J. A. Stankovic, K. Ramamritham and W. H. Kohler, "A Review of Current Research and Critical Issues in Distributed System Software", *IEEE Distributed Processing Technical Committee NEWSLETTER*, vol. 7, pp. 14-47, March 1985.
  - [Stro86] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.
  - [Sun88] Sun Microsystems Inc., "SunOS Users Manual", Mountain View, CA, 1988.
  - [Tane85] A. S. Tanenbaum and R. Van Renesse, "Distributed Operating Systems", *Computing Surveys*, vol. 17, pp. 419-470, December 1985.
  - [Tant85] A. N. Tantawi and D. Towsley, "Optimal Static Load Balancing in Distributed Computer Systems", *Journal of the ACM*, vol. 32, pp. 445-465, April 1985.
  - [Trip88] S. K. Tripathi, D. Finkel and E. Gelenbe, "Load Sharing in Distributed Systems with Failures", *Acta Informatica*, vol. 25, pp. 677-689, June 1988.
  - [Vivas90] V. Vivas, "Design of the Mentat Scheduler", *Masters Thesis, Department of Computer Science*, University of Virginia, May, 1990.



# Performance Evaluation of the Sylvan Multiprocessor Architecture

*F. J. Burkowski*  
*Charles L. A. Clarke*  
*S. C. Cowan<sup>†</sup>*  
*Gordon J. Vreugdenhil*

Multiprocessor Systems Group  
Department of Computer Science  
University of Waterloo  
Waterloo, Ontario N2L 3G1

<sup>†</sup>Currently at University of Western Ontario  
London, Ontario

sylvan@watmsg.waterloo.edu

## Abstract

Sylvan is a multiprocessor system that provides lightweight tasks with full address space protection. The primary design goal of the system is to make tasks viable as a scalable unit of concurrent abstraction. Lightweight tasks are supported by a tasking coprocessor that provides fast context switching and low-latency inter-task communication and synchronization. With fast context switching and communications available, programs can be designed using many concurrent tasks, allowing programs to be distributed across multiple processors, yet still run efficiently on a small number of processors. This paper describes the architecture of the Sylvan system and details performance measurements made on the existing single-node four-processor system. Based on these performance measurements, we discuss the value of the architectural features of the Sylvan system, and describe some additional architectural features which would further enhance performance. These architectural features are evaluated quantitatively where possible.

## 1 Introduction

A network of workstations contains considerable parallel processing power that is not presently being exploited. To address this, several distributed operating systems have been developed that provide an environment suitable to developing parallel distributed applications composed of communicating tasks [10, 17, 21]. Performance, particularly the latency of inter-task communications, is essential to efficient exploitation of the processing resources. These systems provide excellent remote communications latency, and may even be approaching the theoretical limits of their networks. Local communications performance, however, is correctly perceived by developers to be too slow relative to that which is possible in a shared memory environment, and so are dropped in favour of a shared memory model when parallel applications are being developed. Unfortunately, shared memory does not easily or efficiently scale up to be implemented across multiple machines on a network.

To resolve the conflict between scalability and local performance, Sylvan provides hardware support for local inter-task communications, reducing the cost of a request to a server to approximately the cost of a subroutine call. With fast context switching and communications available, programs can be designed using many concurrent tasks so they can be distributed across multiple processors, yet still run efficiently on a small number of processors.

Sylvan is a multiprocessor system that provides hardware support for “tasking” operations, such as communications and scheduling, in order to provide tasks that are both “lightweight” and exist in separate address spaces. Tasks are considered to be “lightweight” if an inter-task communication cycle occurs with latency close to that of a procedure call. Lightweight tasking is realized by supplying a simple core of hardware services which extend standard CPU facilities to encompass task scheduling, context switching, communication, and synchronization. Operating-system services use this hardware core as the basis for their implementation.

A Sylvan task has its own context and thread of execution. When a task is created, it is assigned a unique Task IDentifier (TID) and a protected virtual address space. Tasks communicate among themselves by way of *synchronous message passing*[9]. Groups of tasks are used to provide operating system services using a technique known as *multiprocess structuring*.

Sylvan tasks are intended to be used as a unit of abstraction, hence the emphasis on low-latency tasking operations. Tasks are a concurrent abstraction; there are no concurrency issues within a task, all synchronization is done using the tasking primitives provided. Tasks are a scalable abstraction, because the semantics of the tasking primitives are independent of the proximity of the tasks. No special capabilities are granted to physically nearby tasks; all such optimizations are embodied within the tasking kernel.

The concepts of synchronous message passing and multiprocess structuring were first developed in the Thoth operating system [9], and have proven their value in numerous projects and systems [2, 10, 15, 17, 21, 30, 33]. These projects demonstrate that operating systems may be developed in a natural fashion using a simple tasking kernel as a basis. Synchronous message passing and multiprocess structuring have also been proven to scale for use in multiprocessor and distributed systems [5, 8, 17, 21, 26]. The simplicity and utility of these concepts make them an appropriate basis for the tasking model provided by the Sylvan system. Synchronous message passing has the further advantage of mapping directly onto the model of synchronization and communication used in many concurrent programming languages, including Ada [31], Concurrent C [13], and concurrent C++ [4]. Refer to Gentleman [14] or Cheriton et al. [9] for a detailed description of synchronous message passing and multiprocess structuring.

Sylvan uses a shared bus architecture, in which multiple microprocessors connect to the system bus. At the heart of the architecture is a tasking coprocessor referred to as the *Taskmaster* (or TM). The Taskmaster implements a simple kernel which performs task scheduling and message queuing, and directs all context switching and task interactions.

This hardware-based approach is similar to that taken by Roos [27], however Roos does not share our aim of supporting general purpose computing. Roos focussed specifically on the support of Ada tasking on a single processor, where address space protection is not needed. Ramachandran et al. [23] share our aim of supporting general-purpose computing,



but they did not address the problems associated with providing address space protection and virtual memory; additionally, they failed to achieve comparable levels of performance.

The task model provided by the Sylvan system is described in the next section. Section 3 describes the Sylvan architecture and section 4 provides performance measurements. Section 5 discusses and quantifies the benefits of existing and proposed architectural features that enhance tasking performance. Section 6 compares the performance of Sylvan to similar systems, and section 7 presents our conclusions.

## 2 The Sylvan Task Model

A Sylvan task consists of a single thread of execution in a separate protected address space. The unique TID associated with each task allows the rapid direction of a message to a task. The separate protected address space provides inter-task protection, while the following communications primitives provide fast, safe inter-task communications.

There are three communication primitives which are fundamental to the system:

- |                |  |
|----------------|--|
| <b>Send</b>    | specifies a bounded length message to be transmitted to a specified receiving task. The sending task is blocked waiting for a reply message from the receiving task.   |
| <b>Receive</b> | is issued by a task wishing to accept a message. If a message has already been sent to this task, the message and the sender's identity are returned to the receiver. If no message has been transmitted, the receiver is blocked until a message is sent. |
| <b>Reply</b>   | is issued by the receiving task after a successful <b>Receive</b> . <b>Reply</b> transmits a reply message back to the sender and unblocks it. The replying task then continues its execution.   |

The **Send** primitive requires the sending task to specify a *send queue* number identifying one of 16 queues associated with the receiving task. A send queue effectively allows a type, or name, to be associated with an incoming message. A mask may be specified with the **Receive** primitive which allows a task to receive messages from a subset of the queues. This facility, called *message screening*, allows a task to avoid receiving messages which are of a type which it is temporarily unable to handle. Incoming messages are queued in FIFO order at the specified send queue.

These primitives are similar to those that effect *rendezvous* in Ada. **Send**, **Receive**, and **Reply** correspond to Ada's entry call, **accept**, and **end accept** respectively. A send queue roughly corresponds to an entry of an Ada task. Unlike Ada, these primitives do not suffer from an inability to cope with local delay [18], because replies may be issued in arbitrary order.

A subset of a processor's general purpose registers is used for exchanging data with other tasks. A message to be transferred to another task is loaded into these registers and a request is issued to the Taskmaster. These same message registers are used when receiving incoming messages. Presently eight general purpose registers form this message

register set, making the maximum message length 32 bytes. Experience with synchronous message passing has shown this to be sufficient for most interactions [3, 10].

For communicating larger blocks of data, sylvan provides the `Copy_To` and `Copy_From` primitives. When a task A has sent a message to another task, B, and B has received the message but not replied, then B may copy data either to or from A's address space. This facility is not provided by the Taskmaster, but by the memory management subsystem, which is itself a group of tasks.

Interrupt handling is easily integrated into this communication and synchronization model by mapping interrupts into messages. The `Install_Handler` primitive allows a task to specify a function which is to be called whenever a particular interrupt occurs. When called, the function has access to device registers but cannot access the task's address space. The function is responsible for removing the source of the interrupt and may return up to four bytes of data. This data is then translated into a message and sent to the task which installed the function.

The Sylvan task model does *not* include asynchronous message passing, task migration or shared memory. Asynchronous message passing and task migration are features that frequently lead to slower performance of the tasking primitives[8, 28], and thus threaten the use of tasks as a unit of abstraction. Shared memory among tasks is very fast when the tasks are co-resident, but very slow, or totally unavailable, when they are not. As such, programs that depend on shared memory become non-scalable, threatening the use of tasks as a scalable abstraction.

## 3 The Sylvan Architecture

### 3.1 Hardware Description

As illustrated in Figure 1, a Sylvan node contains one tasking coprocessor (the Taskmaster), and several processor complexes. In the existing implementation there are four processor complexes, each of which consists of a MC68030-based Motorola VM04 VERSABus module and a 4 MB VM13 memory module. The processors communicate with the Taskmaster through the MC68030 coprocessor interface, and invoke Sylvan primitives by executing coprocessor instructions that refer to the Taskmaster. The resulting VERSABus accesses are serialized by the bus arbiter. The VERSABus is only used for coprocessor invocations and message transfers between the processors. The more frequent instruction fetches and data accesses for each processor take place on a private bus (the RAMBus) that exists between each VM04 and its local VM13 memory. This assures a processor of fast access to its local memory. The VM13 memory is dual ported so that data may be transferred over the VERSABus when explicitly requested.

The Taskmaster is a bipolar processor based on the AMD 29116 register ALU and the AMD 2910 microsequencer, and was designed as a microcode engine to run the Sylvan kernel exclusively. The execution unit has a load/store architecture: There are 32 16-bit general purpose registers, and all operations except load and store take only registers as operands. Since it is not intended as a general purpose CPU, the Taskmaster has no interrupt or exception handling hardware.

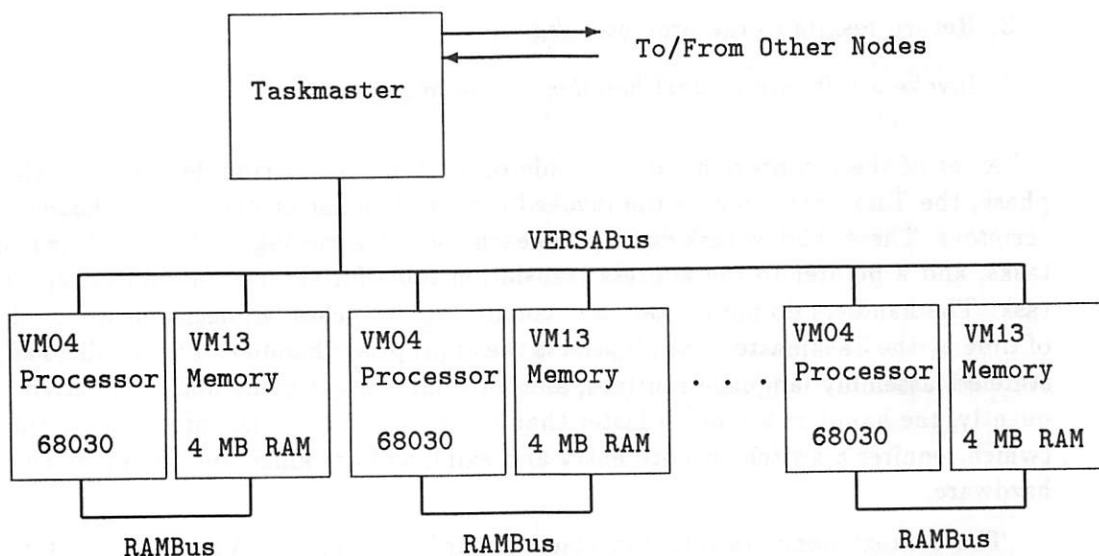


Figure 1: A Sylvan Node

The Taskmaster contains 16 KB of fast static RAM used for internal data structures. Analysis of the Taskmaster microcode [11] shows that two-thirds of the references to local RAM are to fields within a *task descriptor*. A task descriptor is a contiguous 16-word structure aligned to a 16-word boundary, and is used to contain data related to a single task. To allow fast access to fixed fields within a task descriptor, the Taskmaster has a register displacement addressing mode with 4-bit displacement. Since task descriptors are aligned, no addition is used to compute the effective address. Register displacement mode and a register indirect addressing mode are the only addressing modes supported by the Taskmaster.

### 3.2 Kernel Data Structures and Algorithms

The data structures necessary to perform task scheduling and inter-task communications are stored in task descriptors in the Taskmaster's local RAM. The data structures for context switching and message buffering (one message per task) are stored in *shadow task descriptors* [32] located in the memory of the processor complex where the task resides. Partitioning the data structures in this way lowers the cost of the message passing operations. Placing the scheduling information close to the scheduler, and the task context information close to the applications processor, means that hardware units have access to the appropriate data without bus contention.

The processing of a Sylvan primitive involves the following four phases:

1. Read the parameters from the processor registers.
2. Compute state transitions.

3. Return results to the processor registers.
4. Invoke a software *context handler* on the processor.

A set of these context handlers reside on each processor complex. During the fourth phase, the Taskmaster points the invoked context handler at one or two *shadow task descriptors*. These shadow task descriptors each contain a message buffer for their respective tasks, and a pointer to the address translation table for the virtual address space of the task. The handlers do not contain any conditional branches; all decisions are made ahead of time by the Taskmaster when it selects the appropriate handler. The handlers are small, stateless assembly language routines, and have no context to be loaded or saved. Consequently, the handlers are much faster than corresponding entries into a monolithic kernel (which requires a switch on both entry and exit), and are amenable to implementation in hardware.

The context handlers take the place of hardware support for fast context switching on the processors themselves. Such support could take the form of multiple context sets within the processor: multiple banks of registers, and TID-tagging on the cache and TLB entries. We will discuss these ideas further in section 5.

To achieve the smallest possible context switching time with the hardware available, we provide a large number of these handlers. The selection of a context handler specifies one *save* operation, one *action*, and one *load* operation, as follows:

- A **Save** operation specifies the location where the message registers are to be written; one of:

<b>None</b>	no saving necessary
<b>Mine</b>	save message in this task's shadow task descriptor
<b>Other</b>	save message in some other task's shadow task descriptor

- An **Action** specifies the type of action to perform; one of:

<b>Switch</b>	the current task has blocked, context switch to another task
<b>Continue</b>	continue executing the current task

- A **Load** operation designates the location from which to load the message registers; one of:

<b>None</b>	no loading necessary
<b>Mine</b>	load message from this task's shadow task descriptor
<b>Other</b>	load message from some other task's shadow task descriptor

In addition to the general context handlers described above, there are special context handlers that exploit scheduling opportunities:

<b>Idle</b>	there are no other ready tasks; idle the processor
<b>Wake</b>	if same task then resumes, no context switch required



<b>Switch Direct</b>	pass message directly in registers instead of through shadow task descriptors
----------------------	---

The **Idle** and **Wake** handler are applicable only in lightly loaded configurations; the **Switch Direct** handler is frequently applicable.

## 4 Performance

### 4.1 Inter-task Communication Performance

In the algorithms that we have developed, the time overhead of inter-task communication is always bounded above by a constant [5]. The observed time, however, can actually vary by as much as a factor of three, depending on circumstances such as whether the communicating tasks are on the same or different processors, and whether other tasks are ready to run. Here we examine a number of canonical situations that may arise in the execution of a Sylvan program, and detail the time overhead of the tasking operations used.

Measurements accurate to one tenth of a  $\mu$ second were made by tracing the bus transactions between the processors and the Taskmaster on the VERSAbus using a logic probe. Thus the precise time of initiation and completion of all Taskmaster interactions, and trans-processor message movements, can be measured. Behavior of context handlers internal to a single processor are not visible; their times are inferred from the latency to the next Taskmaster request.

To place the following tasking measurements in context, we examined the cost of a subroutine call and return. The subroutine had a 32-byte parameter, and returned a 32-byte result. The latency of the subroutine call and return was 18.1  $\mu$ seconds.

We first examine a test that has two communicating tasks on a single processor with no other ready tasks. Table 1 presents a time analysis for this benchmark, and Figure 2 provides a pseudo-code listing of the test. Each entry in Table 1 gives the time taken by the specified primitive or context handler. Taskmaster primitive entries have been further broken down into times for reading input parameters, computing state changes, and writing out results to the processor.

Task A sends a message to task B, which invokes the first context handler to save the message from registers into A's shadow task descriptor. Task B then executes a **Receive** primitive, which accepts the message from a queue, and uses a context handler to read the message from A's shadow task descriptor into registers. Task B then replies to task A, invoking the **Switch Direct** special context handler. This handler switches directly to task A, leaving the message entirely in registers. Since the last context handler does no work other than to change context from one task to another, we can regard its run time of 32.1  $\mu$ seconds as the time cost of a context switch. Similarly, we can use the difference in run time between the first and third handlers to derive the cost of saving a message to

Processor(s)	Primitive	Handler	Cost $\mu$ seconds	% of Total
CPU & TM	A sends to B		14.3	10.51
CPU		save mine switch load none	46.0	33.82
CPU & TM	B receives from A		10.1	7.43
CPU		save none continue load other	19.6	14.41
CPU & TM	B replies to A		13.9	10.22
CPU		save none switch load none	32.1	23.60
Total Cycle Time			136.0	100.00

Table 1: Communicating Tasks on the Same Processor

registers as 13.9  $\mu$ seconds. Most importantly, we notice that 64.2  $\mu$ seconds or 47% of the total time is spent changing contexts.

We next examine two communicating tasks on different processors. This test is identical to the first test, except that Task A and Task B execute on different processors. Table 2 presents the timing breakdown of this test. With two active processors, some operations take place concurrently. The figures under Critical Path are the contribution of each operation to the total latency overhead, while the figures under Unit Cost represent the latency of the operation on that processor. The Concurrent Operation entry represents the difference between the critical path time, and the sum of the unit costs (latencies) of the operations.

In this test, there is only one task on each processor, so when a task blocks the Taskmaster does not cause a context switch, but instead puts the processor into an idle state (using the **Idle** handler). When the task becomes ready again, the scheduler notices that it is the same task that was running previously, and so again avoids the cost of context switching (using the **Wake** handler). Because the handlers involved in this test never cause context switches, both their unit latency and critical path times are very low, making this the fastest test possible. Breaking down the latency of this test into categories, we find that 43.4  $\mu$ seconds or 44% was spent by the Taskmaster scheduling tasks, while 54.5  $\mu$ seconds or 56% was spent in handlers that do little more than transfer messages.

The third test is identical to the second, except that a lower-priority background task running an infinite loop is present on each processor. This illustrates the message passing cycle time in a heavily loaded setting (each processor is CPU-bound). Every time

```

Task A:
  ttid <- Get_TID(Task B)
  loop forever
    Send(ttid, message)
  endloop
endtask A

Task B:
  loop forever
    ttid <- Receive(message_buffer)
    Reply(ttid, reply_message)
  endloop
endtask B

```

Figure 2: A Pair of Communicating Tasks

one task sends or replies to the other, the Taskmaster detects that a task with higher priority than the task currently running has become ready, and interrupts that processor. This interrupt causes the processor to execute a **Reschedule** primitive. The Taskmaster gives a destination context to the processor during a Reschedule, and a context switch is performed.

Table 3 illustrates the timing breakdown of this test. Again, with two active processors, many operations take place concurrently, so their full cost is not reflected in the Critical Path column. The latency of the cycle time has risen because of the need to interrupt and reschedule a processor on each **Send** and **Reply**. The cost of the context handlers has increased; 131.6  $\mu$ seconds (50%) of the critical path is spent interrupting and rescheduling processors. This cost is an obvious byproduct of pre-emptive scheduling and involuntary context switching; however, during part of this time, the other processor is running the background task, achieving useful work. Under more realistic circumstances, the communicating tasks would do some local computation, giving the background task more time to compute before being interrupted, presumably furthering some other task, and preserving the overall throughput of the system.

## 4.2 Interrupt Response

The test shown in Figure 3 was used to measure the latency of translating interrupts into messages. This test is of particular significance, because it is this mechanism that can be used to respond to outside events such as user key strokes, or the arrival of a disk block. Interrupt processing in traditional systems is typically highly restricted, slow, or both. The Sylvan interrupt handling mechanism is both fast and fully integrated into our tasking model.

This test measures the amount of time between an interrupt event and the start of execution of the appropriate task. Delivery of the interrupt notification message unblocks and schedules this task. During the test, a background task is executing. Table 4 illustrates the chronological components of this test. Interrupt handling breaks down into 43.7

Processor(s)	Primitive	Handler	Critical Path $\mu$ seconds	Unit Cost $\mu$ seconds	% of Total
CPU 0 & TM	A sends to B		18.1	18.1	18.5
CPU 0		save other idle load none	16.0	16.0	16.3
CPU 1		save none wake load mine	12.0	28.0	12.3
CPU 1 & TM	B replies to A		16.3	16.3	16.6
CPU 1		save other continue load none	14.0	17.7	14.3
CPU 0		save none wake load mine	12.5	26.5	12.8
CPU 1 & TM	B receives and blocks		9.0	12.2	9.2
CPU 1		save none idle load none	0.0	16.0	0.0
Cycle Time			97.9	150.8	100.0
Concurrent Operation			52.9		54.0

Table 2: Communicating Tasks on Different Processors

$\mu$ seconds (39%) for actual interrupt processing, 32.5  $\mu$ seconds (29%) to reschedule the processor, and 36.1  $\mu$ seconds (32%) to context switch to the receiver task.

With a total interrupt translation time of 112.3  $\mu$ seconds, a single processor can handle over 4000 interrupts per second, with more than 50% of CPU time still available for real work. This interrupt rate is sufficient to support 4 serial lines running at 9600 baud, or disk blocks arriving at 4000 blocks per second continuously.

## 5 Feature Analysis

Using a coprocessor to execute the tasking kernel provides two first order benefits:

- State save and restore on every entry and exit of the operating system is no longer needed.
- The kernel is the only program running on the Taskmaster so all data structures are ready in registers and local static RAM.

```

Task A:
  /* Install interrupt handling
    function. */
  Install_Handler(vector, handler)

  /* Receive interrupt messages. */
  loop forever
    itid <- Receive(message_buffer)
  endloop
endtask

/* Interrupt handling function. */
handler:
  return(data)
endfunction

/* Background task
  (runs at low priority). */
Task B:
  loop forever
endtask

```

Figure 3: Interrupt Response Test



Processor(s)	Primitive	Handler	Critical Path $\mu$ seconds	Unit Cost $\mu$ seconds	% of Total
CPU 0 & TM	A sends to B		21.8	21.8	8.4
CPU 0		save other switch load mine	0.0	22.8	0.0
CPU 1		Interrupt	40.3	40.3	15.4
CPU 1 & TM	Reschedule		16.2	16.2	6.2
CPU 1		save mine switch load mine	48.3	48.3	18.5
CPU 1 & TM	B replies to A		19.4	19.4	7.4
CPU 1		save other continue load none	0.0	28.9	0.0
CPU 0		Interrupt	39.1	39.1	15.0
CPU 1 & TM	B receives and blocks		8.7	13.3	3.3
CPU 0 & TM	Reschedule		14.2	14.2	5.4
CPU 0		save mine switch load mine	52.9	52.9	20.3
Cycle Time			260.9	317.2	100.0
Concurrent Operation			56.3		21.6

Table 3: Communicating Tasks on Different Processors with a Background Task

These benefits may be measured by comparing a tasking kernel that does not use a coprocessor to a kernel that does. Vadura [32] provides a description of a software only implementation of a kernel that does not provide separate address spaces, but is otherwise similar to the Sylvan kernel. In Burkowski et al. [5], a description is given of a prototype implementation of a Taskmaster-assisted Sylvan kernel that does not provide separate address spaces. Both were executed on Motorola VM04 processors.

The software only kernel of Vadura achieved a message passing cycle time of 232  $\mu$ seconds; the Taskmaster-assisted kernel of Burkowski et al. achieved a message passing cycle of 65  $\mu$ seconds. Not all of the improved performance, however, can be attributed simply to using a coprocessor, many other factors contribute. In the following sections, we detail these factors. We discuss the benefits of the architectural features included in the Sylvan system, and estimate the benefits of future features that may further enhance tasking performance.

Processor(s)	Primitive	Handler	TM Phase	Cost $\mu$ seconds	% of Primitive	% of Total
CPU		interrupt		43.7	100.00	38.9
CPU & TM	Reschedule		input	4.6	14.15	4.1
			compute	9.2	28.31	
			output	18.7	57.54	
			total	32.5	100.00	
CPU		save mine switch load none		36.1	100.00	32.1
Response Time				112.3		100.0

Table 4: Interrupt Response Timing

## 5.1 Taskmaster-specific Features

The Taskmaster's instruction set architecture contains many standard features that contribute to its performance. These were detailed in section 3.1. Performance can be further enhanced by exploiting some simplifying assumptions that can be made.

Much of the difficulty in supporting long pipelines stems from the need for precise exceptions, low-latency interrupt response, and from instructions of different lengths[22, Chapter 6]. Since the Taskmaster is not a general purpose processor, we can exploit the following properties:

- The Taskmaster does not have to support precise exceptions because it never suffers any exceptions.
- The Taskmaster simply responds to requests in FIFO order, it need never be interrupted.
- No floating point is needed for the kernel; all instructions can be of uniform length.

Thus issuing pipelined and superscalar instructions is greatly simplified, and can be used to a much greater extent without checkpointing. The current Taskmaster implementation is not pipelined, but does have nearly uniform instruction lengths, and supports a limited form of superscalar instruction issue; the ALU and instruction sequencer can operate in parallel. The Taskmaster achieves a static clocks-per-instruction ratio of 0.916.

## 5.2 Context Handlers

It is difficult to measure the general improvements due to the use of the context handlers since there is no concrete piece of software with which to compare. Our context handlers are between 5 and 17 instructions long, and average 13 instructions, which is much shorter than the context switching path in conventional operating systems, and is as close as possible to the direct hardware support that we would like.

Tests: Singe CPU Only		TM Compute Time	TM Time	Message Cycle Time
One Pair of Tasks	Time in $\mu$ sec.	18.1	38.3	136
	Compute Cost %	100	47	13
Two Pairs of Tasks	Time in $\mu$ sec.	44.4	113.8	343.3
	Compute Cost %	100	39	13

Table 5: Compute Time Candidates for Concurrent Context Saving

However, we can measure the benefits due to special handlers exploiting scheduling opportunities. The major difference between test 2 and test 3 is that the Taskmaster is prevented from using the **Idle** and **Wake** handlers described in section 3. Considering test 2 as an optimization of test 3, we see that the **Idle/Wake** handlers reduce the message cycle latency from 260.9  $\mu$ seconds to 97.9  $\mu$ seconds, an improvement of 166%. A similar comparison between test 1 and one in which the **Switch Direct** handler is defeated shows a performance improvement of 26%. Thus a decrease in context manipulation can result in a large improvement in overall performance.

### 5.3 Coprocessor Interface

While a significant degree of parallelism is observed in the tests that involve more than one processor (tests 2 and 3), no parallelism at all exists in the single processor tests. This is because the Motorola coprocessor interface does not provide a low-cost asynchronous invocation mechanism of the form:

1. Make request.
2. Do work (save context).
3. Await result (new context to switch to).

Using such a coprocessor interface, context saving could proceed in parallel with scheduling computations in the Taskmaster, hiding the cost of scheduling behind the cost of state saving in operations that lead to context switches. Table 5 illustrates the cost of Taskmaster scheduling computation (Taskmaster Compute Time), showing that the cost of scheduling in the Taskmaster is 13% of the total message cycle time. Reducing this cost to 0 by performing it in parallel would produce an improvement of 15%.

### 5.4 Dedicated Taskmaster

Much of the time in conversation with the Taskmaster is spent passing parameters back and forth between the Taskmaster and the processor. This might suggest that the Taskmaster is too far away, and should be closer to the processors; dedicated to a single processor. Dedicating a Taskmaster to a single processor has two positive effects and two

Tests: Single CPU Only		TM-CPU I/O	TM Time	Message Cycle Time
One Pair of Tasks	Time in $\mu$ sec.	16.8	38.3	136
	I/O Cost %	100	44	12
Two Pairs of Tasks	Time in $\mu$ sec.	48.4	113.8	343.3
	I/O Cost %	100	43	14

Table 6: I/O Cost Fraction of Taskmaster Time

negative effects. The positive effects are: (1) contention for the Taskmaster and the shared bus is eliminated, and (2) the cost of Taskmaster I/O is reduced. The negative effects are: (1) the single-node, multiple CPU case, which Sylvan makes very fast, disappears, and (2) Taskmaster utilization is reduced.

Table 6 illustrates the cost of parameter passing between the Taskmaster and the processor in single processor tests. For these tests, the cost of Taskmaster I/O averages 13%, and eliminating this cost would produce an improvement of 15%.

## 5.5 Multithreaded Processors

The purpose of the context handlers is to assist the Taskmaster in context switching the processors as quickly as possible. A better solution would be to provide hardware support for fast context switching in the processor itself, such as that prototyped in the Denelcor HEP[29]. Some current RISC processors, such as the AMD 29000[16] and the Sun SPARC[12], provide support for multiple contexts in the form of multiple register banks, and TID-tagging of cache and TLB entries. The APRIL project at MIT[1] is investigating the use of multiple register banks in just this way using a modified SPARC processor.

An innovative possibility for multi-threaded processors attached to a Taskmaster is for the Taskmaster to provide a *next context* TID to the processor ahead of time. When a blocking primitive is executed, the processor can switch to the next task immediately, without waiting for the Taskmaster to decide who should go next, effectively pipelining the entire scheduling computation. This restricts the system to scheduling algorithms that are unaffected by the current primitive, i.e. a single level of priority, with a circular queue of ready tasks. For cases where a higher level of priority is essential, the Taskmaster could resort to the interrupt/*Reschedule* technique used when a higher priority task has become ready on a processor that has not issued a primitive.

## 6 Relative Performance

In this section, we briefly compare the performance of Sylvan to similar systems. Sylvan is most similar to the V system[8, 10], since both were derived from Cheriton's Thoth. Like

Sylvan, V provides a tasking model in which tasks are given a separate address space, and communicate by synchronous message passing. In [8], Cheriton reports a local message passing cycle time of 480  $\mu$ seconds for a 32-byte message, in contrast to Sylvan's 136  $\mu$ second message cycle time for the same message size.

The Amoeba system[21] is a minimal tasking kernel similar to V System, but also has capability and object oriented software layered in and on top of the tasking facility. The local remote procedure call time (semantically identical to a message passing cycle) is 800  $\mu$ seconds, again in contrast to Sylvan's time of 136  $\mu$ seconds.

In [24, 25], Ramachandran et al. describe a proposal and simulations for a system very similar to that originally proposed for Sylvan in 1985[6, 7]. Development is presumed to be concurrent, since no reference is made to the earlier Sylvan proposal. Ramachandran presents two significant systems (labeled II and III) running on Motorola VERSAmodule Monoboard Microcomputers[20] (an 8 MHz MC68000 version of the VM04). System II is essentially similar to Sylvan in its hardware makeup, and system III describes a (simulated) system with a custom bus enhanced to support queue manipulations. System II is reported to have an SRR latency of 5598  $\mu$ seconds, and system III a latency of 3912  $\mu$ seconds.

While the 68000 processors used are approximately three times slower than Sylvan's 68030 processors, the performance figures are 30 times slower than the Sylvan times. Furthermore, the performance of these systems is five times poorer than that of the V system reported in[10] on comparable 68000 processors, without hardware assistance. Rather than enhancing tasking performance, Ramachandran's hardware support seems to have *slowed down* tasking operations. On examining the internal structure of Ramachandran's system, we discover that some of this cost is due to his decision to place *all* data for task descriptors in a memory that is shared between the processor and his tasking coprocessor. Thus the critically important scheduling and context switching information is far away from both the processor and the scheduler, slowing down all tasking operations by 17%.

Recently, two new efforts have been undertaken to further reduce the cost of inter-task communications through software-only systems. In [19], Massalin and Pu describe the Synthesis Kernel, which dynamically generates small fragments of code to perform specific tasking operations, similar to Sylvan's context handlers. The performance achieved by Synthesis is excellent (under 20  $\mu$ seconds for a context switch), but has not incorporated multiple processors or virtual memory, which add to complexity and degrade latency. Massalin and Pu also do not discuss the semantics of inter-task communications facilities, so a direct comparison to the message passing cycle cannot be made.

In [3], Bershad et al describe a mechanism called "Lightweight Remote Procedure Call." LRPC is said to be a combination of the "heavy-weight" semantics of remote procedure call (message passing) between separate threads, and the lighter-weight semantics of capability-protected simple procedure call. The essential aspect of normal RPC that has been removed for performance reasons is the existence of a separate thread for servers. This simplification has lead to performance gains: the null LRPC call costs 127-157  $\mu$ seconds (depending on optimizations). However, the LRPC semantics sacrifice the existence of the server's thread, essentially rendering a LRPC server as a monitor.



## 7 Conclusion

The Sylvan system provides tasks that are viable as a unit of abstraction by enhancing the latency of tasking operations above what is normally possible. Using such tasks as units of abstraction allows modular systems to be constructed that are both responsive to external input (interrupts), and are scalable. Responsiveness is preserved by integrating the interrupt mechanism into the low latency tasking operations. Scalability is preserved by providing semantics in tasking primitives that are independent of proximity.

## 8 Acknowledgments

The authors would like to express their sincere appreciation to Dennis Vadura for the time which he spent helping us with the details of this paper. We would also like to thank Gordon Cormack, John Rogers and Dermot Harriss for their comments and suggestions.

## References

- [1] Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiawicz. APRIL: A Processor Architecture for Multiprocessing. *Computer Architecture News*, 18(2):104–114, June 1990. Proc. 17th International Symposium on Computer Architecture.
- [2] Richard J. Beach, John C. Beatty, Kellogg S. Booth, Darlene A. Plebon, and Eugene L. Fiume. The Message is the Medium: Multiprocess Structuring of an Interactive Paint Program. *Computer Graphics*, 16(3):277–287, July 1982.
- [3] Brian N Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight Remote Procedure Call. In *Proceedings of the Symposium on Operating Systems Principles*, 1989.
- [4] P. A. Buhr, Glen Ditchfield, and C. R. Zarnke. Adding concurrency to a statically type-safe object-oriented programming language. *SIGPLAN Notices*, 24(4):18–21, April 1989. Proc. ACM SIGPLAN Workshop on Object-Based Concurrent Programming.
- [5] F. J. Burkowski, G. V. Cormack, and G. D. P. Dueck. Architectural Support for Synchronous Task Communication. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–53, Boston, MA, April 1989.
- [6] F. J. Burkowski, G. V. Cormack, J. D. Dymont, and J. K. Pachl. A Message-Based Architecture for High Concurrency. In *Proc. 1st Conference on Hypercube Multiprocessors*, pages 27–37, Knoxville, Tennessee, August 1985.
- [7] F. J. Burkowski and D. Dymont. Sylvan: A Message-Based Multiprocessor Architecture. In *Proc. 18th Hawaii International Conference On Systems Sciences*, pages 31–42, Honolulu, January 1985.

- [8] David R. Cheriton. The V Distributed System. *Communications of the ACM*, 31(3):314-333, March 1988.
- [9] David R. Cheriton, M. A. Malcolm, L. S. Melen, and G. R. Sager. Thoth, A Portable Real-Time Operating System. *Communications of the ACM*, 22(2):105-115, February 1979.
- [10] David R. Cheriton and W. Zwaenepoel. The Distributed V Kernel and its Performance for Diskless Workstations. In *Proc. 9th ACM Symp. Operating Systems Principles*, pages 128-140, October 1983.
- [11] S. Crispin Cowan. Realization and Analysis of the Sylvan Kernel. Master's thesis, University of Waterloo, 1990.
- [12] R. Garner, A. Agarwal, F. Briggs, E. Brown, D. Hough, B. Joy, S. Kleiman, S. Munchnik, M. Namjoo, D. Patterson, J. Pendleton, and R. Tuck. Scaleable Processor Architecture (SPARC). In *COMPCON, IEEE*, San Francisco, California, 1988.
- [13] N. H. Gehani and W. D. Roome. Concurrent C. *Software — Practice and Experience*, 16(9):821-844, September 1986.
- [14] W. Morven Gentleman. Message Passing Between Sequential Processes: the Reply Primitive and the Administrator Concept. *Software — Practice and Experience*, 11:435-466, 1981.
- [15] W. Morven Gentleman. Using the Harmony Operating System. Technical Report NRCC No. 23030, National Research Council of Canada, December 1983.
- [16] Mike Johnson. *Am29000 Streamlined Instruction Processor*. Advanced Micro Devices, 1988.
- [17] Frank Kolnick. *The QNX Operating System*. Basis Computer Systems, Inc., Willowdale, Ont., 1989.
- [18] B. Liskov, M. Herlihy, and L. Gilbert. Limitations of Synchronous Communication with Static Process Structure in Languages for Distributed Computing. In *13th Annual ACM Symp. on Principles of Programming Languages*, pages 150-159, June 1986.
- [19] Henry Massalin and Calton Pu. Threads and Input/Output in the Synthesis Kernel. In *Proceedings of the Symposium on Operating Systems Principles*, 1989.
- [20] Motorola, Inc. *VERSAmodule Monoboard Microcomputer User's Guide*, 1982.
- [21] S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse, and H. van Staveren. Amoeba — A distributed Operating System for the 1990's. *IEEE Computer*, 23(5), May 1990.
- [22] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, first edition, 1990.

- [23] Umakishore Ramachandran, Marvin Solomon, and Mary Vernon. Hardware Support for Interprocess Communication. *Computer Architecture News*, 15(2):178-188, June 1987. Proc. 14th International Symposium on Computer Architecture.
- [24] Umakishore Ramachandran, Marvin Solomon, and Mary Vernon. Hardware Support for Interprocess Communication - II. Technical Report GIT-ICS-88/13, Georgia Institute of Technology, 1988.
- [25] Umakishore Ramachandran, Marvin Solomon, and Mary Vernon. Hardware Support for Interprocess Communication. *IEEE Transactions on Parallel and Distributed Systems*, 1(3), July 1990.
- [26] John W. Rogers. Distributing Sylvan: Distributed Kernel Primitives for a Fast Message-Passing Multiprocessor. Master's thesis, University of Waterloo, 1990.
- [27] J. Roos. A Real-Time Support Processor for Ada Tasking. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 162-171, Boston, MA, April 1989.
- [28] M. L. Scott and A. L. Cox. An Emperical Study of Message-Passing Overhead. In *7th International Conference on Distributed Computing Systems*, pages 536-543, Berlin, W. Germany, September 1987.
- [29] Burton J. Smith. A Pipelined, Shared Resource MIMD Computer. In *Proceedings of the International Conference on Parallel Processing*, 1978.
- [30] F. Tuynman and L. O. Hertzberger. A Distributed Real-Time Operating System. *Software — Practice and Experience*, 16(5), May 1986.
- [31] United States Department of Defence. *Reference Manual for the Ada Programming Language ANSI/MIL-STD-1815A-1983*. United States Department of Defence, February 1983.
- [32] Dennis Vadura. A Proposed Design for the Sylvan Kernel. Master's thesis, University of Waterloo, 1985.
- [33] Waterloo Microsystems, Inc., Waterloo, Ont. *Waterloo Port User's Guide*, 1984.



# Debugging Multiprocessor Operating System Kernels \*

Noemi Paciorek  
noemi@encore.com

Susan LoVerso  
sue@encore.com

Alan Langerman  
alan@encore.com

Mach Operating System Group  
Encore Computer Corporation  
257 Cedar Hill Street  
Marlborough, MA 01752

## Abstract

Encore has developed several Unix based multiprocessing operating systems since 1984. We have accumulated considerable experience implementing and debugging parallelized operating systems. Kernel debugging in general is sometimes considered a mysterious process. Debugging multiprocessor kernels seems even more arcane. In this paper, we shed some light on multiprocessor debugging. We focus on general debugging tools and techniques with special attention on locking problems. We give some examples drawn from our most recent experiences with Encore Mach and OSF/1.

## 1 Introduction

Unix operating systems traditionally target uniprocessor hardware. All of the major time-sharing dialects inherited the original uniprocessor Unix assumption of no kernel-mode preemption. Consequently, synchronization between competing activities in a Unix kernel usually focuses on preventing races between interrupt-level code and process-context code that share data structures. The Unix kernel enforces this synchronization by disabling interrupts at one of several possible levels to eliminate conflicting interrupt-level activities. Unfortunately, the interrupt-level synchronization model does not generalize well to symmetric shared-memory multiprocessor (SMP) architectures.

Typical SMP hardware has separate interrupt inputs for each processor. Most SMP implementations have no way in hardware to mask all processors' interrupt levels simultaneously. In fact, such an ability would be undesirable because it would prevent the machine from processing interrupts at a rate significantly faster than that of a uniprocessor. A few machines restrict most I/O to a single processor but even in these cases all processors can take *some* interrupts, if only an inter-processor interrupt or a time-slice-end. Thus, blocking an interrupt level on a single processor in

\*This research was supported in part by the Defense Advanced Research Projects Agency (DoD) through ARPA Order No. 5875, monitored by Space and Naval Warfare Systems Command under Contract No. N00039-86-C-0158.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Multimax, UMAX4.3 and UMAXV are trademarks of Encore Computer Corporation. Unix is a trademark of Unix System Laboratories. OSF/1 is a trademark of the Open Software Foundation. NFS is a trademark of Sun Microsystems, Inc.



such a machine does not ensure that some other processor will not be interrupted at the same level. Therefore, interrupt exclusion alone does not guarantee correct synchronization in a general-purpose multiprocessor operating system.

To preserve the traditional Unix kernel synchronization model, the operating system can bind the execution of all kernel code to a single processor, effectively forcing what might otherwise be symmetric multiprocessor hardware into a master/slave configuration. User-level programs can be allowed to run on any available processor; user-level programs have always themselves been preemptible, so presumably they have already been coded to anticipate and prevent races with other user-level programs. This technique permits the operating system to continue using interrupt masking as its synchronization model while distributing user-level computation across all available processors. Unsurprisingly, running the kernel in a master/slave paradigm degrades the system's performance. Only one of possibly many processors can be applied towards kernel-mode processing but Unix can spend considerable time in the kernel. Ironically, if the operating system also schedules user processes on the designated "master" processor, kernel requests generated from other processors might be forced to wait until the program using the master relinquishes its quantum.

A more sophisticated approach to multiprocessor synchronization uses semaphores or locks to synchronize simultaneous contention for kernel resources or data structures. The use of locks in operating systems has long historical precedent[5]. Indeed, even Unix uses locks[11] on occasion to control access to data structures when a process might sleep before completing its activities. In contrast, a lock-based kernel associates a lock with every data structure that could be used by more than one processor simultaneously. By convention, the kernel acquires the appropriate lock each time it examines or modifies a shared data structure. Depending on the data structure and the algorithms manipulating it, the lock might be a binary semaphore (also known as a "mutual exclusion" lock), a multiple-reader/single-writer lock, or some other synchronization primitive. If the lock isn't immediately available, the kernel forces the process requesting the lock to busy-wait or to sleep until the lock becomes available. Thus, locks synchronize the effects of competing processors on shared data structures.

Of course, a lock-based kernel must still use interrupt masking on occasion to avoid corrupting data structures shared between process-context and interrupt-context. Interrupt-level activities in a multiprocessor kernel can be synchronized in one of two ways. First, interrupt-level activity can be reduced in scope greatly by enqueueing appropriate information about the interrupt to a kernel-mode process dedicated to handling interrupts. In this way, most interrupt-level synchronization problems in the operating system become problems of coordinating competing processes, solvable by applying locking techniques.

Alternately, on a multiprocessor, locks can be associated with interrupt levels, permitting interrupt activity to synchronize directly with process-context activity. The locks used must not be "blocking", or sleepable locks, for the obvious reason that in interrupt-context the kernel has no process to suspend. By sharing locks and data structures between interrupt-level and process-context, the latency and extra context switch of using dedicated interrupt-handling threads can be eliminated. On the other hand, the implementation might become significantly more complex. Process-context code might become slower because every use of a lock shared with interrupt-level code must be guarded by appropriate interrupt masking. All lock-based kernels make tradeoffs between using dedicated threads and using direct, interrupt-masked locks to synchronize interrupt-level activity.

The application of lock-based synchronization techniques to Unix kernels remains uncommon. A few vendors have been supporting proprietary versions of lock-based versions of Unix for a number of years[6, 13, 9], some based on a semaphored version of Unix released by AT&T in 1984[1]. The Open Software Foundation has just released a version of Unix that uses locks for its fundamental synchronization primitives. OSF/1 is based on Mach, an operating system written from scratch at Carnegie-Mellon University for distributed and multiprocessor systems, with further parallelization enhancements by Encore. However, traditional commercial offerings such as System V Release 4,

SunOS and Xenix continue to employ interrupt exclusion as the primary kernel synchronization technique. (System V and SunOS are being rewritten to use locks; an upcoming System V release incorporates some fine-grained locking with minimal algorithmic modifications[8].) Experiences with debugging multiprocessor Unix kernels therefore are not widespread and this lack of experience fosters the perception that debugging multiprocessor operating systems is a mysterious art.

In truth, multiprocessor operating systems can be more frustrating to debug than uniprocessor operating systems simply because the activities many processors must be tracked. However, many techniques from traditional uniprocessor kernel debugging remain useful; in some cases, those techniques must be generalized. In the past 7 years, Encore has developed or enhanced the multiprocessing capabilities of four Unix-like operating systems:

- Umax4.2/4.3, derived from Berkeley sources
- UmaxV, based originally on System V Release 2
- Encore Mach Release 1.0, based on Carnegie-Mellon University Mach Release 2.5
- OSF/1, based on CMU Mach Release 2.5 and Encore Mach Release 1.0

We have gained considerable experience with the debugging of lock-based Unix kernels. We will discuss some of the techniques we have developed for debugging these operating systems and our experiences using them in the context of Encore Mach and OSF/1[4, 10, 12]. However, the techniques and examples presented have equal applicability to both Encore Mach and OSF/1 and generally we will not distinguish between them.

We assume the reader has a basic familiarity with the internals of the Unix operating system. Note, however, that Mach internals differ substantially from Unix internals. In particular, Mach offers tasks and threads instead of processes (a single task containing a single thread emulates a Unix process). We will employ the terms “process-context” and “thread-context” interchangeably.

In Section 2 we briefly describe the synchronization primitives used in Mach and OSF/1 to give a concrete example for the rest of the paper. We detail the trade-offs of applying a lock hierarchy to prevent deadlocks in Section 3. We give an overview in Section 4 of the tools we use to debug our multiprocessor operating system kernels: an interactive source-level kernel debugger and a post-mortem crash dump analyzer. Section 5 relates techniques we use to debug locks and problems that arise from incorrect lock usage. One of the most useful techniques we use, assertions, is analyzed in Section 6. Problems with incorrectly setting processor priority levels cause as many, if not more, problems on a multiprocessor than on a uniprocessor; we discuss some of these problems in Section 7. We conclude with a brief summary.

## 2 Synchronization

The Mach kernel provides one fundamental synchronization primitive, the *simple lock*, which implements an abstraction of hardware-assisted mutual exclusion. Typically, the *simple\_lock()* function uses a non-blocking, atomic test-and-set instruction when attempting to acquire the lock. If the test-and-set fails, *simple\_lock* spins in a tight loop waiting for the lock to be released, at which time the function tries the test-and-set again. Because of the spinning action, these locks are also known generically as “spin locks”. Mach assumes that simple locks are an inexpensive synchronization mechanism: they are used directly in many sections of code to guard data structures. More complex synchronization mechanisms may be constructed on top of the simple lock primitive.

Mach also provides a multiple-reader/single-writer lock abstraction, for brevity’s sake dubbed a *read/write lock*. The read/write lock permits multiple readers to hold the lock at the same time;

but the lock serializes writers both with respect to each other and with respect to the readers. The read/write lock finds frequent use in situations where multiple threads of control can share non-destructive access to a data structure but only one thread at a time can be permitted to modify the data structure. As implied above, Mach employs a simple lock to serialize access to the read/write lock data structures.

Interestingly, the Mach read/write lock can assume both blocking and non-blocking modes. In the vast majority of cases, the kernel uses blocking read/write locks: failure to acquire a read/write lock causes the thread to be suspended until the lock becomes available. In the non-blocking mode, however, processors attempting to acquire the lock will loop repeatedly until the lock becomes available. Also, a non-blocking lock can be used by interrupt-level activities as well as within thread context. Use of non-blocking read/write locks is rare. In the rest of this paper, we will assume that "read/write lock" means "blocking read/write lock".

We must mention three additional features of the Mach lock package. First, rather than provide a separate interface for mutual exclusion (*mutex*) locks, the caller of the lock package uses a read/write lock but always acquires it for writing. By following this convention, only one thread at a time ever acquires the lock.

Second, the Mach lock package supplies *conditional* locking interfaces for both simple locks and read/write locks. If the lock is available, the conditional lock routine acquires it. If the lock *cannot* be acquired immediately, without spinning or sleeping, the conditional lock routine returns an error code to the caller. The caller then assumes the burden of taking further action, such as releasing locks already held before attempting to unconditionally reacquire the desired lock. Avoiding deadlock frequently requires using conditional locking (see Section 3).

Finally, a read/write lock may be marked *recursive*, when necessary. In this case, the thread owning the lock may reacquire it without first releasing it. Normally, of course, two successive lock attempts by the same thread on the same lock leads to deadlock on the second lock attempt. Lock recursion is a convenience more than a necessity, occasionally permitting the implementor to re-use code without having to re-design locking strategies or re-implement interfaces.

Other synchronization primitives may be built on top of simple locks, but to date we have not observed a great need for others. At Encore, we have also built an event package (post, clear, test and wait on an event) and interlocked queues but in practice we employ these synchronizers rarely relative to vanilla simple locks and read/write locks.

## 3 Lock Hierarchies

### 3.1 A Hierarchy

As the number of locks in the operating system increases, the possibility of deadlock increases exponentially. Some developers use a global lock hierarchy to organize their locks to guarantee that deadlock will never happen.

An global lock hierarchy avoids deadlock by ordering all lock acquisitions according to a "lock level". A lock may be acquired only if its level is greater than that of all locks currently held. Following this rule guarantees that deadlock will never result because locks are always acquired in the same order. Generally, the locks may be released in any order without affecting the deadlock-avoidance properties of the hierarchy.

A lock hierarchy also has the useful feature that the hierarchy rules can be checked easily at run-time. Each type of lock (*e.g.*, a socket lock) is assigned to one of the levels in the hierarchy. When

acquiring a lock it is only necessary to verify that its level exceeds that of all the locks currently held. Verification can be as simple as checking a bitmask representing the levels of the currently held locks. (Typically, this checking code is only compiled into the kernel when the kernel is built with some kind of debugging option.) In a typical lock hierarchy implementation we would keep two bitmasks, one for simple locks and one for read/write locks. Each process has its own read/write lock bitmask because read/write locks can be held across blocking operations that cause the process to move from one processor to another. The simple lock bitmask, on the other hand, has one copy per processor because simple locks may never be held across blocking operations.

It is important to note that the problems uncovered by these checks are not necessarily actual run-time deadlocks but rather *potential* deadlocks. Of course, in some situations acquiring locks in reverse-level order may be entirely safe despite the hierarchy violation. If the hierarchy checking package complains about such a situation, the algorithm must be rewritten to conform with the hierarchy's constraints or the checking package must somehow be told to stop complaining. It must also be understood that a run-time hierarchy checking package cannot detect hierarchy violations in unexercised code paths, so the use of a hierarchy does not obviate the need for thorough test coverage.

At the risk of some false positives, a lock hierarchy checking package theoretically will catch many of the potential deadlocks, even ones that might otherwise be difficult to discover through stress testing. Thus, a lock hierarchy is advocated on the grounds that it produces structured locking that also can easily be checked at run-time for potential deadlocks.

In fact, we used a lock hierarchy when we developed our first parallelized operating system, Umax4.2. Umax4.2 was designed with a lock hierarchy in mind, to guarantee correctness of the resulting implementation. Although in a strict hierarchical design all lock accesses conform to the hierarchy, the Umax4.2 designers also permitted the use of conditional locking to evade the hierarchy's restrictions. However, use of conditional locking was rare because the designers applied the hierarchy rigorously.

The lock checking features of the Umax4.2 hierarchy package revealed many simple deadlock cases early in the kernel development cycle. After the operating system began to experience regular internal use, however, further improvements in system reliability resulted not so much from hierarchy checking but from the application of a systematic quality assurance effort.

Umax4.2 was largely rewritten from its origins in 4.2BSD. Primary motivations for the rewrite were the need to add multiprocessor support and the desire to better structure the kernel implementation. Part of the reason to restructure the kernel stemmed from the fact that, as originally written by Berkeley, the algorithms and data structures did not lend themselves to the use of a hierarchy. The Berkeley code assumed a uniprocessor environment, in which most kernel state is available whenever needed and kernel resources can be used in nearly any order. Some cases exist wherein the 4.2BSD kernel obeys locking and ordering restrictions but these cases are the exception rather than the rule. Thus, in the effort to apply locks to the Umax4.2 kernel, much of the kernel was restructured or rewritten, in part to obey the lock hierarchy.

However, no one anticipated the need to upgrade the operating system quickly to new levels of functionality or to add third-party system software. For example, even the simplest changes between 4.2BSD and 4.3BSD were difficult to import into Umax4.2 because the structure of the Umax4.2 internals was so different from that of 4.2BSD. Adding NFS functionality became an unusually lengthy task because the Sun reference sources were derived from 4.2BSD and Umax4.2 internals had little commonality with 4.2BSD.



### 3.2 A Generalized Protocol

A strict lock hierarchy actually is a proper subset of all lock protocols. A general lock protocol can take or release locks in accordance with its knowledge of the data structures at hand. Locks can be acquired in different orders depending on specific knowledge of the algorithms being used. Of course, like a lock hierarchy, a lock protocol must always prevent deadlock. (In fact, a hierarchy with an evasion feature, such as conditional locking, can express any general lock protocol. Technically speaking, a lock hierarchy that permits evasion isn't really a lock hierarchy.) For example, a typical lock order may be:

1. acquire lock A
2. acquire lock B.

However, situations may arise in which the locks must be taken in the reverse order:

1. acquire lock B
2. acquire lock A.

When the second situation arises, we must be careful to avoid deadlock: one processor could be executing the first code sequence while another executes the second. We can avoid the deadlock by conditionally acquiring lock A, as follows:

1. acquire lock B
2. conditionally acquire lock A (*breaking a hierarchy's rules*)
3. if successful, done
4. if unable to acquire lock A:
  - (a) release lock B (*B's resource temporarily unprotected*)
  - (b) acquire lock A
  - (c) acquire lock B
  - (d) re-check state A and B's resources, if necessary

This algorithm brings out two important points. First, the use of conditional locking itself breaks a strict hierarchy's rules, as we have already mentioned. The algorithm above, therefore, can only be part of a general lock protocol and cannot be legal with a lock hierarchy. If conditional locking is used frequently, a hierarchy may not have much value, either for design or for run-time checking.

Second, if the conditional lock attempt fails the algorithm must carefully release and re-acquire the locks in the correct order. Worse, releasing lock B opens up a window during which time some other processor might modify the state associated with lock B. Usually the algorithm must re-check relevant state after successfully acquiring locks A and B to ensure that the original conditions applying to the operation in progress remain valid. However, in a strict hierarchical protocol we would *always* have to follow these steps of releasing and re-acquiring locks followed by re-checking. The only other choice in a strict lock hierarchy is to rewrite the larger algorithm so that locks A and B are always acquired in the correct order. Using a general protocol, the algorithm above may frequently avoid the penalty of unlocking, relocking and re-checking state.

Another method of avoiding the deadlock is using some special knowledge about the algorithms or data structures involved to guarantee that deadlock will not result from acquiring the locks in reverse order. For example,



1. acquire lock B
2. /\* recognize special case, permitting reverse order acquisition \*/
3. acquire lock A

We have two operating systems that do not rely on lock hierarchies. One, UmaxV, was implemented soon after Umax4.2. Based on System V Release 2, it was already largely parallelized by AT&T for the multiprocessor 3B2 base. While Encore performed considerable work enhancing single-stream and parallel performance, adding functionality and fixing bugs, it was not deemed worthwhile to add a lock hierarchy to the operating system. To add one would have required significant recoding to conform with a hierarchical ordering or the extensive use of conditional locking to evade hierarchy checking. In the former case, tracking future releases of System V would have become more difficult; in the latter case, the hierarchy would have had little or no worth as a design or debugging aid.

When we embarked on our Mach effort, we noticed that the reliability of our UmaxV product was quite good despite its lack of a lock hierarchy. We observed that the use of a hierarchy in Umax4.2 did not reveal all lock problems immediately; substantial and heavy stress testing was still required. We also noticed that UmaxV was far more amenable to the incorporation of third-party system software and upgrades than was Umax4.2. We decided that we would parallelize the Unix pieces of Mach with as few modifications as possible, to permit us to easily track future developments in Mach and 4.3BSD. For all those reasons, we preferred to use general lock protocols that could be tailored to existing algorithms and data structures. We discovered that with the addition of a simple lock-checking package, described in Section 5, we were able to detect most of our locking problems early in the kernel development cycle (just as if we had used a lock hierarchy). Those lock problems not detected during development revealed themselves during stress testing, as with Umax4.2/4.3 and UmaxV.

Our experiences demonstrate that producing a successful lock-based operating system does not automatically require the use of a global lock hierarchy. Obviously, many relationships within a general lock protocol will be hierarchical; but by avoiding the temptation to impose a global ordering on all locks the need to restructure code (or evade the hierarchy) will be reduced. The addition of a simple lock checking package permits lock problems in a general protocol to be caught as early as if a hierarchy was used. General lock protocols better fit existing code and data structures, yielding long-term benefits when maintaining the kernel and tracking outside developments.

## 4 Debugging Tools

Debugging multiprocessor operating systems, as with uniprocessor operating systems, can be a difficult and time consuming process. We have observed several stages of debugging:

1. Early in the boot sequence. The kernel cannot assist its own debugging yet via breakpoints or crash dumps. Solution of problems at this stage may require additional hardware assistance (*e.g.*, an in-circuit emulator or a logic analyzer) or the use of *printf*s.
2. Kernel-assisted debugging. The kernel is sufficiently stable that it can be single-stepped under its own power. Often the developer is interested in a problem confined to a single module, which can be analyzed by tracing the execution of the code. These problems frequently are traditional uniprocessor problems, although it is useful from time to time to attack multiprocessor problems in this fashion.
3. Autopsies. The kernel crashes, or is induced to crash, so that its remains may be studied with a post-mortem analyzer. Complex problems that may be difficult to reproduce are often best

attacked with a crash dump analyzer. Problems resulting from multiprocessor race conditions tend to be most easily solved via autopsy; interactively debugging many processors can be quite tedious.

In fact, these stages apply equally well to uniprocessor and multiprocessor debugging. The multiprocessor case has the additional complication that more than one processor can be implicated in a bug, so it must be possible to examine each processor in the system.

Carefully designed tools can greatly reduce debugging time in the second and third stages. We detail our methods for interactive multiprocessor debugging and give a brief description of our post-mortem analyzer.

## 4.1 Hardware Base

Before describing our interactive debugging techniques, we digress briefly to describe our hardware platform. Encore manufactures a tightly-coupled, symmetrical shared-memory multiprocessor called the Multimax[2]. The machine supports up to 32 processors per box. All processors can use all main memory and I/O can be initiated and completed by any processor. Hardware maintains cache coherency across all processors.

## 4.2 Interactive Debugging

The Multimax provides a small amount of hardware support for kernel debugging. Each processor card has a dedicated serial port for the interactive debugging of operating system and stand-alone software. Commands sent over that port are received and interpreted by a nearly stand-alone debug monitor incorporated into the kernel. The debug monitor, *dbmon*, was developed at Encore using the National Semiconductor NS32000 ROM monitor as a model. *Dbmon* understands commands to dump and modify registers and memory, single-step the processor, and manipulate the state of the memory management unit.

*Dbmon* contains a small state machine that indicates which processor “owns” *dbmon*. The processor owning *dbmon* is responsible for carrying out any commands that arrive over the debug serial line. The boot processor performs I/O over the serial line on behalf of the processor that actually owns *dbmon*. All other processors in the system idle while *dbmon* is active. Commands from the machine’s console can be used to interrupt *dbmon* and force it to switch ownership between processors. Thus, we can interactively examine and set the state of any processor in the system.

While commands can be typed directly to *dbmon* over the debug serial port, typically we use a source-level debugger instead. Our standard user-level source debugger is Third Eye Software’s *cdb*[7]. It offers a variety of standard debugging features, such as statement single-stepping and formatted data structure display. In 1986, Encore modified *cdb* slightly to produce *rdp*, our remote kernel debugger. These modifications consisted of small changes to the portion of *cdb* responsible for fetching data from the target core file. Instead of issuing commands to read the core file, *rdp* sends synchronous queries over the debug serial line to be executed by *dbmon*. Nearly for free, we get the full power of an interactive source debugger applied to kernel debugging.

*Rdp* actually knows nothing about processors; neither the processor it is manipulating nor the other processors in the system. As mentioned above, all knowledge about which processor answers *rdp*’s requests resides in *dbmon*. *Rdp* always debugs “the current process” and only *dbmon* knows that the process is actually a *processor*. *Rdp* blithely sends its requests to the boot processor’s debug line and prints the results according to the responses it receives.

As shown in Figure 1, we usually start a few windows on an X terminal to control the debugging

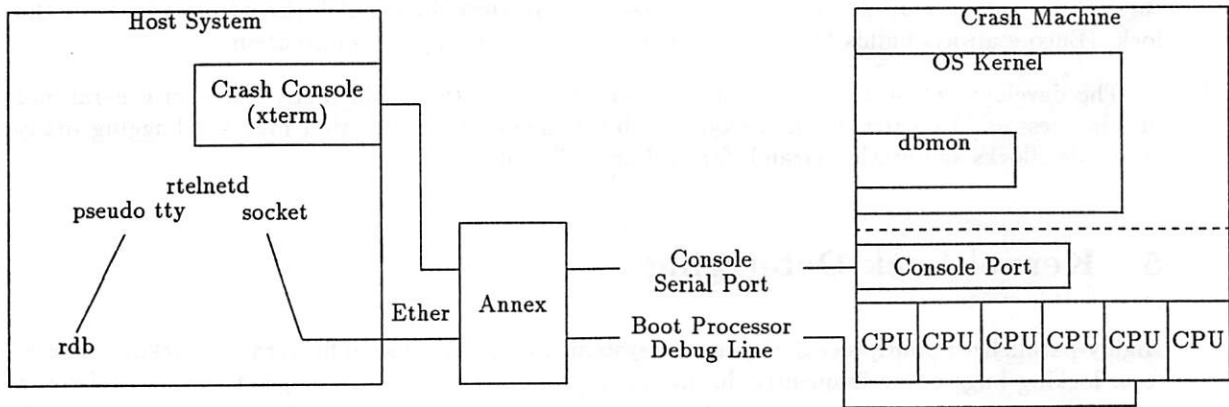


Figure 1: Debugging a Remote Multiprocessor

process. Rdb runs on a host system, separate from the target machine. We run rdb in one window and open up an xterm window to the crash machine's console port, for direct control over the crash machine. Rdb itself communicates with the remote system over a tty. Since there are no serial lines on the Multimax, the *rtelnetd* program serves as an intermediary between a pty (used by rdb) and a socket to the terminal server, which has the actual debug serial line. One valuable side-effect of this scheme is that we can do the actual debugging from our office or even dialed-in from home, while still maintaining complete control over the crash machine.

Rdb is very useful during the early stages of kernel debugging and when focusing on specific problems that do not involve large amounts of kernel state. When the kernel crashes or hits a breakpoint, we can connect rdb to dbmon and debug the crash remotely. Rdb allows us to dump any memory address and to disassemble instructions. We can also obtain stack backtraces or examine the state of locks or any other kernel data structures. If the kernel has been compiled with extra symbol information for debugging, rdb can be used to display data structures symbolically. Otherwise, rdb can be used to dump relevant memory areas for further analysis.

Unfortunately, rdb has a few shortcomings. When the kernel has not been compiled for debugging, we cannot display data structures symbolically. Rdb has no way to analyze the state of multiple processors without using console commands to dbmon, which makes frequent examination of the state of all the processors in the system quite tedious. Rdb also does not have a sophisticated notion of data formatting and of the ways we like to group data structures together.

### 4.3 Post Mortem Tool

Our post mortem tool is *mda*, the Mach crash Dump Analyzer. Mda was developed at Encore to run on Mach based systems. It owes a debt to previous work done at Encore on Umax4.2 and UmaxV crash dump analyzers. Mda can be thought of as a traditional crash dump analyzer extended to recognize an array of processor data structures. Mda maintains a notion of the current processor to ease the user's job. Where appropriate, mda knows about all the processors in the crash dump.

Mda allows us to trace threads, as well as to symbolically display the contents of thread, task, and other kernel data structures. It can display all the threads active on each processor when the crash occurred or all the threads in the system. Mda also allows stack backtraces per thread, as well as per processor.

Mda displays information from related data structures grouped together and pays special atten-

tion to the contents of kernel locks. Using mda, we can easily trace deadlocks; mda reveals which threads are spinning or sleeping on a lock and then in turn show which thread already owns the lock. (Encore always builds Mach kernels with extra lock debugging information.)

The development of a crash dump analyzer that understands Mach structures in general and multiprocessors in particular has become an invaluable tool for us. Mda makes debugging many kernel deadlocks and crashes straightforward and efficient.

## 5 Kernel Lock Debugging

Highly-parallelized multiprocessor operating systems make heavy use of fine-grained locking. Therefore, locking bugs occur frequently during an implementation: the developer forgets to release a lock, or takes the same lock twice, or takes locks in the wrong order. As we have already mentioned, a lock hierarchy checking package provides mechanisms for run-time checking of common locking problems. However, we chose not to use a lock hierarchy in Mach for the reasons enumerated in Section 3; instead, we use general lock protocols. Of course, we still want run-time checking of our locks so we developed a simple but effective debugging package.

### 5.1 Lock Types

When initializing a lock, the kernel assigns a type identifier to it. Especially when debugging a kernel lacking source code symbol table information, the type unambiguously indicates what kind of lock is being analyzed. The kernel also uses the lock type when recording statistics about lock usage, but a treatment of lock statistics lies beyond the scope of this paper. Our post-mortem crash dump analyzer understands and takes advantage of the lock type.

### 5.2 Post-Mortem Lock Debugging

The lock type does not tell us everything we would like to know when debugging a lock problem. Some locks are taken at over a hundred different places in the kernel. To sort out deadlock problems, we added debugging information to each lock. This debugging information applies to both simple locks as well as read/write locks.

We added the following pieces of data to each lock:

- the *address of the thread* that last acquired the lock
- the *program counter (pc)* at which the lock was last acquired
- the *pc* at which the kernel last released the lock

When analyzing a crash dump, we look at the threads that were actually executing to see whether any are spinning on a simple lock. If we find one or more such threads, we print out the simple lock involved to see what thread actually owns the lock. We immediately know where the owning thread acquired the simple lock and, by tracing that thread's stack, we know its current state. If the owning thread is also waiting on a lock, we repeat this process until we find the source of the deadlock. Here is an example.

In the Virtual File System (VFS), the vnode contains a simple lock used to serialize operations on the vnode's contents. Similarly, the Unix File System (UFS) uses a simple lock to guard the

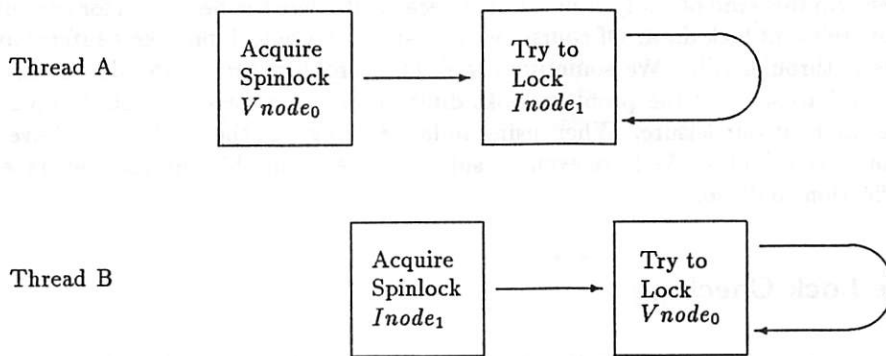


Figure 2: Two Threads In Deadlock

contents of each inode. From time to time, UFS must examine data in the inode and its associated vnode at the same time. There is a precedence between the inode and vnode simple locks: the inode lock must always be acquired first. More generally, the simple lock protecting the filesystem-specific counterpart of the vnode takes precedence over the vnode simple lock. This rule makes sense because the VFS layer *never* manipulates filesystem-specific data. Following the rule prevents deadlock between inode and vnode locks.

Now, consider two threads executing in UFS. Thread A calls *ufs\_badfunc()*, an incorrectly coded routine that first acquires the vnode simple lock and then attempts to acquire the inode simple lock (refer to Figure 2). Simultaneously, thread B, executing in the correctly-coded routine *ufs\_goodfunc()*, acquires the inode lock and then spins on the vnode lock. Deadlock results.

Because the locks in question are simple locks, two processors in the system have been knocked out of action; the rest of the system doubtlessly will deadlock shortly thereafter. We inspect the state of the hung kernel as we described above:

1. We use mda to inspect the threads running on the system's processors.
2. We discover thread B spinning on a lock.
3. Examining thread B's lock, we find it is a vnode lock, currently locked by thread A from *ufs\_badfunc()*.
4. We trace thread A and find it spinning on another lock.
5. Examining thread A's lock, we find it is an inode lock, currently locked by thread B from *ufs\_goodfunc()*.
6. The deadlock between inode and vnode lock is obvious; our knowledge of the locking protocol indicates that *ufs\_badfunc()* is the culprit.

Analyzing a deadlock involving blocking locks can be a bit more difficult because many threads can legitimately sleep on blocking locks simultaneously. We print out a summary of all the threads in the crash dump, then sort the summary by the thread's wait event. If we find many threads waiting for an event that happens to be a lock address, we investigate that lock to determine what thread owns it. As with a simple lock problem, we use the lock debugging data to trace the deadlocked threads.



Typically, we perform this kind of analysis using mda because it offers the best tools for examining and summarizing the relevant lock data. Of course, we can also analyze lock problems interactively, albeit more tediously, through rdb. We sometimes look at a problem first with rdb to discover whether it “looks easy” to solve; if the problem looks difficult, we then force a crash dump so we can study the issue later, at our leisure. When using mda, carrying out the analysis we have just described takes about five minutes. We have even considered automating this analysis but have not had sufficient justification to do so.

### 5.3 Run-time Lock Checking

We also concentrated on adding as much run-time verification of lock state to the kernel as we could. If any of the checks fail, the kernel panics so that the problem can be examined before its effects have a chance to spread. This additional checking is expensive so it is enabled by a configurable option.

The thread owner information saved with each lock for post-mortem analysis also can be used by our run-time checking package. We can detect an attempt by a thread to acquire a lock it already owns, or to release a lock it never acquired. Within the lock package, we targeted two strategic functions for checking lock state. One of these functions, *lock\_write*, acquires a read/write lock for writing. The other, *lock\_done*, releases a read/write lock regardless of how it was acquired.

When acquiring a lock for writing, we usually want to verify that the current thread does not already hold this lock. Before acquiring the lock, *lock\_write* first checks whether recursive locking has been enabled on the lock. If the lock isn't recursive, *lock\_write* checks that the current thread does not already own the lock. Obviously, if the current thread attempts to lock a non-recursive lock it already owns, it will deadlock with itself.

*Lock\_done* also handles the case of a lock supposedly held for writing. First, *lock\_done* checks that the lock really is writelocked. (A common bug is calling *lock\_done* on a lock that was never acquired for writing, or even never acquired at all.) If the first check passes, *lock\_done* confirms that the thread unlocking the lock is the one that originally locked it. Generally, unlocking a lock owned by another thread creates a future nasty surprise for the thread that thought it owned the lock. The OSF/1 and Encore Mach kernels do contain a couple of cases where we deliberately pass locks from one thread to another; in those instances, we use a bit of extra (debugging-only) code to inform the lock package of the change in ownership.

Besides the lock ownership checks, we added counts to detect illegal uses of locks. There are two sets of data recorded when the kernel is configured for debugging. The kernel maintains a per-processor count of spin locks held and a list of their addresses. The kernel also keeps a count of blocking read/write locks held and a list of their addresses on a per-thread basis.

Several strategic points in the code contain checks on these counts and lock addresses:

1. On entry and exit from *syscall()*, no locks of any kind can be held.
2. When a thread blocks, no spin locks can be held.
3. When handling a trap, the same locks must be held on exit as were held on entry.

The first two items are obvious. No thread should ever hold a kernel lock while executing in user space. Checks to make sure the lock counts are zero on entry and exit to *syscall* guarantee that we detect this case. Also, since spin locks cannot be held across blocking operations, we check the count of spin locks before blocking to ensure none are held. Blocking while holding a spin lock introduces the potential for starving processors waiting for the lock or even deadlocking the entire system.

The third item is rather interesting. For the first two cases, we could have simply maintained the lock counts. However, when we exit a trap, we want to guarantee not only that the number of locks held has not changed, but also that the same locks are held as when the trap was taken. We need to record not only how many locks are held, but also the identities of those locks. We save the counts and locks held on entry to a trap. Before returning from the trap, the thread's saved lock counts and addresses are compared with the current ones, to verify that they match.

The thread ownership and lock count checks seem straightforward, but we encountered two interesting implementation problems. We naively incremented the counts in the lock functions, and decremented the counts in the unlock functions. However, there are several places where a free data structure is taken from a pool of such structures, initialized, used and returned to the pool. We discovered a few places where these structures are returned to the pool still locked. This is, technically, a perfectly legitimate use of the lock and the structure. The structure will be reinitialized by the next thread to fetch it from the pool. Using structures in this manner triggered our lock count checks. In these cases, we added extra debugging code to unlock the structure before returning it to the pool.

As we mentioned earlier, the lock package believes that a lock should be released by the thread that originally acquired it. One instance which deliberately violates this assumption is the buffer cache code, where the thread initiating asynchronous I/O is generally not the same thread that signals the completion of the I/O and releases the buffer's lock. (The buffer is protected by a blocking lock held across the I/O operation.) For debugging purposes, the lock ownership passes from the initiating thread to the thread that completes the I/O. There is a window during the actual I/O when the lock does not have an owner, although it remains locked. Immediately prior to beginning the I/O, the initiating thread relinquishes its ownership of the lock. However, it cannot know what thread will signal the completion of that I/O. When the I/O finishes, the thread signalling the completion will first acquire ownership of the buffer lock before any use of the buffer itself. This way ownership of the buffer can be passed, but we do not break any of our checks on buffer lock ownership. We must also modify the lock count information for each thread involved in this operation.

Experience has shown that gathering the thread ownership information costs little and greatly aids debugging, so we collect this information even in production kernels. The additional run-time lock ownership and lock count checking takes a substantial amount of time, so we do not include these checks in production kernels. We find that these mechanisms detect most lock-related problems early in the kernel development cycle.

## 6 Assertions

An assertion statement provides a "sanity" checkpoint and guarantees state information assumptions are true. If a thread or data structure is found to be in an unexpected state, the machine panics. Assertions can check any state information, (e.g. if the current thread holds a particular lock or if the current thread is executing on a particular processor). Any time an assertion fails, a fundamental assumption about the operating system's state at that time has been violated and the kernel panics. We can then use a crash dump analyzer to investigate the problem.

We implemented the lock state checking described in Section 5.3 as assertions. We also use assertions outside of the lock debugging package to verify the state of locks and kernel data structures. Some functions expect to be passed locked objects while others expect to be passed unlocked objects that will be locked later. We use assertions in both of these cases to ensure that the locks on data structures are in the state we expect them to be in. For instance, the Unix File System (UFS) uses assertions to verify the state of inode locks. The inode read/write blocking lock is held for a relatively long periods of time and often across function calls. Many functions in the UFS layer

expect to be passed locked inodes. Routines expecting to be passed an inode locked for writing can assert that the thread currently executing holds the inode read/write lock for writing. These assertions have been useful in complex pieces of code where locks are held across function calls and may be acquired and released by different routines.

We also use assertions to verify that kernel data structures themselves are in the state the kernel expects. The Virtual File System (VFS) uses them to check the state of vnodes. For example, when a vnode is no longer used and is being freed back into the pool of unused vnodes, we assert that only the current thread has a reference to that vnode. Similarly, when referencing a vnode we assert that the vnode is not a member of the free pool.

We also use assertions to verify the state of many other kernel data structures. Certain system calls must execute on a particular processor and during the course of those calls, we frequently assert that the current thread remains on that processor. We use assertions to verify the validity of pointers to various data structures.

Our experience with assertions has shown that they are a very useful kernel debugging tool. They are extremely helpful in the early stages of kernel development and testing. However, they do increase the system overhead, especially if placed in critical code paths, and they are unnecessary in stable, debugged kernels. Therefore, assertions are a configuration option that may be turned off, alleviating the overhead in stable kernels. This is in contrast to standard Unix panics which always perform checks for conditions that rarely, if ever, happen. Occasionally, we still use panics to detect unrecoverable error conditions but generally we favor the use of assertions over panics. Assertions also enable us to detect many problems earlier when they may be easier to understand and solve, before they reveal themselves in potentially strange ways.

## 7 Interrupt State Debugging

In Mach, a given lock must be acquired at the same interrupt priority level. Failure to do so may result in a system deadlock or hang. We record various data about the interrupt state of the processors to assist us in debugging hung kernels resulting from acquiring locks at inconsistent interrupt levels. The interrupt debugging assertions also catch failures to re-enable interrupts once they have been disabled.

The interrupt debugging code saves the following information for each processor:

- the *address of the last thread* interrupted
- the *pc* at which the thread was interrupted
- the *type* of the last interrupt
- the *pc* at which interrupts were last enabled
- the *pc* at which interrupts were last disabled

We use this information in conjunction with the lock debugging package to pinpoint locks that are acquired at inconsistent interrupt priority levels. The following scenarios describe problems that result from acquiring locks at incorrect interrupt levels and how we use the lock and interrupt debugging code to solve them.

## 7.1 Simple Interrupt-State Deadlock

Any lock that may be acquired in interrupt context must always be acquired at an elevated interrupt level. Whenever the lock is acquired in thread context, any interrupts that also may attempt to acquire the lock must be disabled. Suppose a thread attempts to acquire a lock that is also acquired in interrupt context without disabling interrupts. If the thread is interrupted and the interrupt handler tries to acquire the lock, the handler will spin waiting to obtain the lock. But the interrupt handler will spin forever trying to acquire the lock because the lock is held by the thread it interrupted.

The crash dump obtained from this hang shows one or more processors spinning trying to acquire the lock. The lock debugging package provides information about the thread owning the lock and the address where it was locked. The saved interrupt level debugging information shows that interrupts were enabled when the thread acquired the lock. This is also obvious from the fact that an interrupt occurred but the interrupt debugging information also reveals where interrupts were last toggled. This condition can be quickly debugged and fixed by always acquiring the lock in thread context with the correct interrupts disabled.

## 7.2 “TLB Shootdown” Deadlock

A more complex system deadlock results when a lock is acquired at inconsistent interrupt levels and a TLB shutdown occurs. Each processor has a translation lookaside buffer (TLB) that caches page table entries. Whenever a page table entry (PTE) is updated, all the processors that cached the entry must flush their TLBs. Mach accomplishes this flushing by performing a TLB shutdown. A detailed description of the TLB shutdown code is beyond the scope of the paper, but is described in [3]. However, a brief description is necessary before describing deadlocks that may occur during a TLB shutdown. Here is a brief description of the TLB shutdown algorithm:

1. A processor determines that it needs to update one or more PTEs.
2. The processor invalidates its TLB.
3. The processor sends a TLB shutdown interrupt to each processor that may have cached the PTEs signaling them to invalidate their TLBs.
4. Each processor receiving the interrupt sends an acknowledgement back to the signaling processor.
5. Each processor receiving the interrupt then waits for the signaling processor to complete the synchronization with all the processors.
6. The signaling processor waits for all the acknowledgments.
7. After all the acknowledgments have been received, each processor receiving the interrupt flushes its TLB.

When locks are acquired at inconsistent interrupt levels, the system may deadlock during a TLB shutdown. Suppose two threads attempt to acquire the same spin lock but one thread disables interrupts before acquiring the lock. The following scenario will lead to a system hang as shown in Figure 3:

- Processor A acquires the spin lock with interrupts enabled.
- Processor B attempts to acquire the spin lock with interrupts disabled.
- Processor C initiates a TLB shutdown.

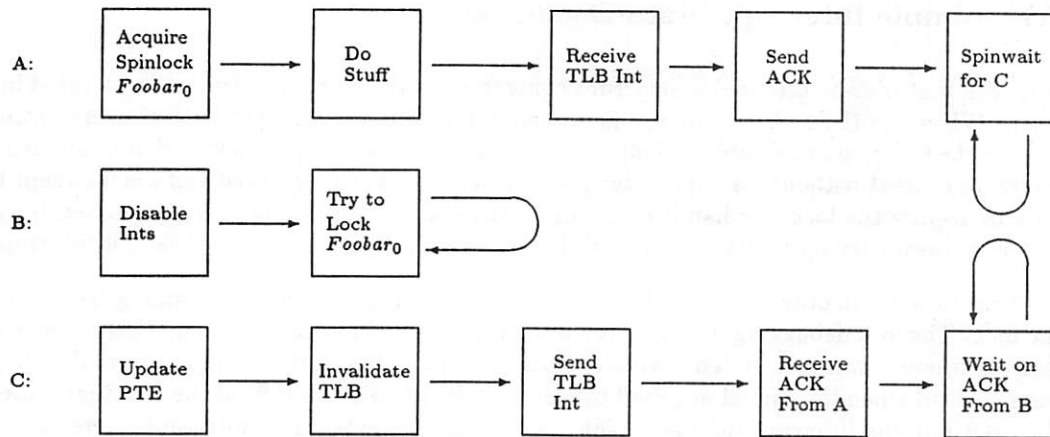


Figure 3: TLB Shootdown Deadlock

We can use the lock and interrupt state debugging information saved to determine that this hang is caused by acquiring locks at inconsistent interrupt levels. We debug a crash dump for this hang as follows:

1. We trace all the processors and see that all but processors B and C are spinning waiting for processor C to complete the synchronization with all the processors.
2. We trace processor C and find it waiting for an acknowledgment from processor B.
3. We trace processor B and find it spinning on a lock.
4. The lock debugging information shows that processor A owns the lock processor B waits to acquire.
5. The interrupt state debugging information shows that processor A has interrupts enabled while processor B has interrupts disabled.
6. The interrupt state debugging information also shows the pc where interrupts were disabled on processor B.
7. We generally fix this problem by applying one of the following techniques:
  - If the lock is never acquired in interrupt context, we always take it with interrupts enabled.
  - If the lock may be acquired in interrupt context, we always disable interrupts before acquiring the lock in thread context.

So either processor A should have interrupts disabled or processor B should have interrupts enabled.

Without the interrupt state and lock debugging information we saved, this hang would have been more difficult to debug. The interrupt state and lock debugging packages greatly reduced debugging time and also greatly simplify debugging this kind of deadlock.

## 8 Summary

Debugging multiprocessor systems can be made easier by using appropriate conventions and tools. When using an evolving source base, we advocate the use of general lock protocols rather than a



global lock hierarchy. The locking problems flagged by a lock hierarchy can also be solved easily with a lock debugging package. Such a package must provide information for run-time checking and post-mortem analysis. Traditional interactive and post-mortem debugging tools must be extended to understand multiprocessor systems. Being able to verify interrupt state also turns out to be extremely useful, as locks sometimes become intimately entwined with interrupt levels.

Kernel debugging has always been a mysterious art, but taken as a whole debugging multiprocessor kernels need not be much more difficult than debugging uniprocessor kernels.

## References

- [1] M. Bach and S. Buroff. Multiprocessor UNIX Operating Systems. *AT&T Bell Laboratories Technical Journal*, 63:1733-1749, October 1984.
- [2] C. G. Bell. Multi: A new class of multiprocessor computers. *Science*, 228, April 1985.
- [3] D. L. Black, R. F. Rashid, D. B. Golub, C. R. Hill, and R. V. Baron. Translation Lookaside Buffer Consistency: A Software Approach. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*, pages 113-122, April 1989.
- [4] J. Boykin and A. Langerman. The Parallelization of Mach/4.3BSD: Design Philosophy and Performance Analysis. In *Workshop Proceedings, USENIX Workshop on Experiences with Distributed and Multiprocessor Systems*, pages 105-126, El Toro, CA, 1989. USENIX Association.
- [5] Harvey M. Deitel. *An Introduction to Operating Systems*. Addison-Wesley Publishing Company, 1983.
- [6] Encore Computer Corporation, Marlborough, MA 01752-3089. *UMAX 4.2 Programmer's Reference Manual*.
- [7] Encore Computer Corporation, Marlborough, MA 01752-3089. *CDB User's Guide*, 1987. Part No. 724-04593.
- [8] M. Campbell et. al. The Parallelization of UNIX System V Release 4.0. In *Conference Proceedings, 1991 Winter USENIX Technical Conference*, pages 307-323, Berkeley, CA, 1991. USENIX Association.
- [9] G. Hamilton and D. Code. An Experimental Symmetric Multiprocessor Ultrix Kernel. In *Conference Proceedings, 1988 Winter USENIX Technical Conference*, Berkeley, CA, 1988. USENIX Association.
- [10] A. Langerman, J. Boykin, S. LoVerso, and S. Mangalat. A Highly-Parallelized Mach-based Vnode Filesystem. In *Conference Proceedings, 1990 Winter USENIX Technical Conference*, pages 297-312, El Toro, CA, 1990. USENIX Association.
- [11] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley Publishing Company, 1989.
- [12] S. LoVerso, N. Paciorek, A. Langerman, and G. Feinberg. The OSF/1 Unix Filesystem (UFS). In *Conference Proceedings, 1991 Winter USENIX Technical Conference*, pages 207-218, Berkeley, CA, 1991. USENIX Association.
- [13] Sequent Inc., Beaverton Oregon. *Balance Guide to Parallel Programming*, 1986.

The first part of the paper discusses the importance of the system architecture in the design of a distributed system. It is argued that the system architecture is a key factor in determining the performance, reliability, and scalability of the system. The paper then presents a detailed description of the system architecture, including the hardware and software components, and the interconnections between them.

The second part of the paper describes the design of the system software. It discusses the various modules and their interactions, and the algorithms used to implement the system's functionality. The paper also presents a detailed description of the system's performance, including the results of various tests and benchmarks.

The third part of the paper discusses the system's reliability and scalability. It presents a detailed description of the system's fault tolerance mechanisms, and the results of various tests and benchmarks. The paper also discusses the system's scalability, and the results of various tests and benchmarks.

The fourth part of the paper discusses the system's security. It presents a detailed description of the system's security mechanisms, and the results of various tests and benchmarks. The paper also discusses the system's security, and the results of various tests and benchmarks.

The fifth part of the paper discusses the system's performance. It presents a detailed description of the system's performance, and the results of various tests and benchmarks. The paper also discusses the system's performance, and the results of various tests and benchmarks.

The sixth part of the paper discusses the system's reliability. It presents a detailed description of the system's reliability, and the results of various tests and benchmarks. The paper also discusses the system's reliability, and the results of various tests and benchmarks.

The seventh part of the paper discusses the system's scalability. It presents a detailed description of the system's scalability, and the results of various tests and benchmarks. The paper also discusses the system's scalability, and the results of various tests and benchmarks.

The eighth part of the paper discusses the system's security. It presents a detailed description of the system's security, and the results of various tests and benchmarks. The paper also discusses the system's security, and the results of various tests and benchmarks.

The ninth part of the paper discusses the system's performance. It presents a detailed description of the system's performance, and the results of various tests and benchmarks. The paper also discusses the system's performance, and the results of various tests and benchmarks.

The tenth part of the paper discusses the system's reliability. It presents a detailed description of the system's reliability, and the results of various tests and benchmarks. The paper also discusses the system's reliability, and the results of various tests and benchmarks.

The eleventh part of the paper discusses the system's scalability. It presents a detailed description of the system's scalability, and the results of various tests and benchmarks. The paper also discusses the system's scalability, and the results of various tests and benchmarks.

The twelfth part of the paper discusses the system's security. It presents a detailed description of the system's security, and the results of various tests and benchmarks. The paper also discusses the system's security, and the results of various tests and benchmarks.

The thirteenth part of the paper discusses the system's performance. It presents a detailed description of the system's performance, and the results of various tests and benchmarks. The paper also discusses the system's performance, and the results of various tests and benchmarks.

The fourteenth part of the paper discusses the system's reliability. It presents a detailed description of the system's reliability, and the results of various tests and benchmarks. The paper also discusses the system's reliability, and the results of various tests and benchmarks.

The fifteenth part of the paper discusses the system's scalability. It presents a detailed description of the system's scalability, and the results of various tests and benchmarks. The paper also discusses the system's scalability, and the results of various tests and benchmarks.

The sixteenth part of the paper discusses the system's security. It presents a detailed description of the system's security, and the results of various tests and benchmarks. The paper also discusses the system's security, and the results of various tests and benchmarks.

The seventeenth part of the paper discusses the system's performance. It presents a detailed description of the system's performance, and the results of various tests and benchmarks. The paper also discusses the system's performance, and the results of various tests and benchmarks.

The eighteenth part of the paper discusses the system's reliability. It presents a detailed description of the system's reliability, and the results of various tests and benchmarks. The paper also discusses the system's reliability, and the results of various tests and benchmarks.

The nineteenth part of the paper discusses the system's scalability. It presents a detailed description of the system's scalability, and the results of various tests and benchmarks. The paper also discusses the system's scalability, and the results of various tests and benchmarks.

The twentieth part of the paper discusses the system's security. It presents a detailed description of the system's security, and the results of various tests and benchmarks. The paper also discusses the system's security, and the results of various tests and benchmarks.

# Debugging the Time Warp Operating System and Its Application Programs

Peter L. Reiher  
Jet Propulsion  
Laboratory  
4800 Oak Grove Drive  
Pasadena, CA 91109  
reiher@onyx.jpl.nasa.gov

Steven Bellenot  
The Florida State  
University  
Tallahassee, FL 32306  
bellenot@math.fsu.edu

David Jefferson  
UCLA  
Los Angeles, CA 90024  
jefferson@lanai.cs.ucla.edu

## Abstract

The Time Warp Operating System (TWOS) runs discrete event simulations on parallel hardware using an optimistic synchronization method based on rollback and message cancellation. Developing this system caused many difficult debugging problems, both because of its unique method of operation and general problems of developing a distributed system. This paper describes some of the techniques used to debug TWOS. These techniques include debuggers built into the operating system, logging methods, graphical tools, internal statistics, special-purpose applications, and monitors. In addition, TWOS has an important property that aids in debugging – simulations run under TWOS must produce deterministic results from run to run. The paper discusses how this property proved useful for debugging both TWOS and the applications run on it.

## 1. Introduction

The Time Warp Operating System (TWOS) is a special purpose operating system designed to run discrete event simulations in parallel with the primary goal of maximum speedup. (TWOS runs simulations; it is not itself a simulation, but a genuine operating system.) It uses an unusual synchronization mechanism based on the theory of virtual time. Every event in the simulation is assigned a virtual time by the user, and TWOS guarantees that the resulting parallel execution will produce results identical to running every event in increasing virtual time order. TWOS actually runs events at many different virtual times in parallel, to provide good speedup. Instead of using a conservative method to determine precisely which events can safely be run in parallel without compromising the sequentiality constraint, however, TWOS permits each node of a parallel machine to run the earliest event it has. As a result, some events may be performed out of order, in which case TWOS must roll back the misordered computation and cancel any of its effects, before going on to process events in the correct order.

Even this brief description of TWOS makes clear that such a system will have substantial debugging problems. Not only is TWOS an operating system, and a distributed operating system, but any action it takes on behalf of the user

may turn out to be erroneous and need to be automatically corrected by the system. In completed, correct runs, only correct actions were taken, but when bugs are present during a run, a person debugging TWOS is faced with a mixture of correct information and incorrect information, with no easy way to distinguish them.

At its inception, the TWOS project had no access to any existing parallel or distributed debugging tools. Such tools did not really exist outside some laboratories, at the time, and certainly did not exist for the experimental hardware used by the project. Nonetheless, little thought was given, at the outset, to the type of debugging support necessary to complete the project. As a result, tools had to be developed as the need arose, rather than in parallel with the systems development.

Despite its novel aspects, the most fundamental principle used in debugging TWOS is familiar to all: get as much information as possible about the problem. Most of the tools discussed in this paper are meant to do exactly that. They offer windows into the behavior of the operating system and the application that allow the debugger to examine as much of the available data as possible. Some of them also allow the debugger to summarize vast quantities of data into a graphical form that is easier to understand. Typically, this data cannot be directly scanned in any other useful way, because of its volume.

One aspect of TWOS debugging is quite unusual, however. Despite presenting an asynchronous synchronization model to its users, TWOS is committed to producing deterministic results. While this goal may sound contradictory with the synchronization method used by TWOS, it actually meshes quite well. TWOS' synchronization method will always guarantee the appearance of a given application's events being processed in exactly one order. With more appropriate hardware and compilers, TWOS' synchronization mechanism would guarantee this ordering without any user intervention. Currently, following certain simple rules when writing the application guarantees that TWOS will always produce exactly the same results from one run to the next. For instance, users are restricted in certain ways in using pointer values.

Determinism has an important implication for debugging. Deterministic programs will always demonstrate the same problems on every run, substantially simplifying the problem of tracking down the error. Moreover, any non-repeatable error is a sure signal of a bug in the operating system itself, assuming that the application does follow the rules.

Most of the debugging methods described here are not completely novel, or are very specific to the problems of TWOS. The main value of this paper is its chronicle of the techniques and tools we found useful in developing a complicated distributed system, rather than as pure research in the field of

debugging. [Cheung 90] describes a more general framework for debugging distributed programs. [Lehr 89] and [Socha 88] describe two actual integrated debugging systems for distributed programs.

## 2. The Time Warp Operating System

A simulation to be run on TWOS must be decomposed into *objects*, which run *events* and send timestamped messages to other objects. (TWOS objects are similar to processes in most operating systems, and can be read as synonymous with "process".) The timestamp on a message is the simulation time at which it is to arrive at its destination object. The arrival of a message at an object causes that object to execute an event at the simulation arrival time. Objects communicate with one another solely by passing messages, with no shared memory whatsoever. Any object may send a message to any other object at any time, without needing to set up any kind of channel between the two objects. Except for initialization and termination code, all user code runs as part of an event.

TWOS runs one simulation at a time, with the goal of completing that simulation as quickly as possible. Each node of the parallel processor hosts several objects, scheduling them independently of all other nodes. The TWOS scheduler always chooses the local object with an unprocessed message at the earliest simulation time to run next. Objects are only preempted when another object receives a message at an earlier time than that of the event currently running.

Since each processor schedules without waiting for, or consulting with, other processors, at any given instant of real time the system's processors may be working at a wide range of simulation times. An object running at a low simulation time can send a message to another object at a higher simulation time. If the message is scheduled to arrive at a simulation time earlier than the receiving object is currently handling, the receiver must *roll back* his computation to the time of the newly arrived message. Any erroneous work done by the out-of-order computation must be totally undone. Undoing the erroneous work requires throwing away local results and sending message cancellations to other objects. TWOS is able to correctly undo any work done prematurely, along with any side effects it may have had. Rollback and message cancellation are totally transparent to the application program.

Every object has a set of private variables called its *state*, which cannot be directly examined by any other object. Every event causes the creation of a new version of the state, timestamped with the simulation time of the event. TWOS typically keeps multiple copies of each object's state in order to support rollback.

At any given moment in a TWOS run, the simulation's objects have performed some work correctly, and some work in error. TWOS periodically



calculates the earliest simulation time that could still be in error. Any work done for simulation times earlier than that time will never be rolled back, and can be *committed*. Both events and messages can be committed. A *committed message* is one that would have been sent in the sequential run of the program, and a *committed event* is one that would have been performed in the sequential run of the program. In essence, these committed actions represent the correct path of computation for a simulation. To meet its definition of correct behavior, any event or message TWOS commits must exactly correspond to an event or message that would be committed in a sequential run of the same simulation, and every message or event in a sequential run of the simulation must be matched by a message or event in the committed trace of the parallel run. Committing a message means that that message buffer can be freed. Committing an event means that the associated state can be discarded.

TWOS periodically runs a calculation to determine which messages and events can be considered committed. Essentially, anything earlier than the earliest unprocessed event will never be rolled back. The simulation time of that earliest unprocessed event is called global virtual time, or GVT. TWOS calculates a conservative estimate of GVT so that it can free storage used by committed messages and events that need no longer be saved.

The TWOS project has developed a sequential simulator called TWSIM that runs exactly the same simulations as TWOS. TWSIM is a conventional event list simulation engine designed to support application prototyping and provide single processor performance figures. TWSIM is a primary tool for debugging new applications. Since TWOS is committed to producing deterministic results precisely the same as those of TWSIM, any application bug present in a TWSIM run would also cause a problem under TWOS. Users can thus do much debugging sequentially, which is substantially easier. TWSIM uses a central event queue implemented as a splay tree, and has been extensively optimized for speed. It runs on one processor of the same hardware as TWOS itself. The sequential simulator never does work optimistically, and never needs to roll back any work it has done.

Since user code will sometimes execute optimistically down incorrect paths, TWOS must be prepared to handle all kinds of errors. User code that would operate correctly if given the proper message sequence (as it would get under TWSIM) can often fail when given misordered messages. If TWOS were completely and correctly implemented, it would be able to handle any such problems, including addressing exceptions, floating point exceptions, division by zero, and even infinite loops. Events causing such problems would be marked as erroneous. If they were rolled back, the error would be ignored. If they were committed, then the user has written genuinely erroneous code that will fail either sequentially or in parallel, and TWOS would flag the error. TWOS is not yet complete, so it does not always deal with exceptions

properly. Certain user errors are already caught and marked, demonstrating that the basic method of handling these problems works.

Experience with TWOS has shown that optimistic execution can provide excellent speedup of discrete event simulation, despite fairly frequent rollbacks. TWOS has achieved speedups in excess of 40 times the speed of the same simulation performed by TWSIM [Hontalas 89].

This description of TWOS is necessarily brief, and does not cover the theory of virtual time that underlies its operation [Jefferson 85], nor many important and interesting details of its implementation [Jefferson 87]. Several other implementations of virtual time synchronized distributed simulation systems also exist, and methods of performing distributed simulations in totally different ways have been developed [Fujimoto 90].

TWOS has been under development at the Jet Propulsion Laboratory since 1983. It has been a complete, functional system since 1986. TWOS has run on a variety of parallel and distributed architectures, including the Caltech/JPL Mark 2 Hypercube, the Caltech/JPL Mark 3 Hypercube, the BBN Butterfly GP1000, and networks of Sun3 and Sun4 workstations connected by an Ethernet.

### 3. Debugging and Determinism

The value of providing deterministic results for debugging parallel and distributed systems is widely recognized [Socha 88], [Lin 88]. However, providing determinism for all runs (not just debugging replays) on a system supporting an asynchronous model of user communications is not easy [Emrath 88]. None the less, TWOS must provide deterministic results to its users on all runs [Reiher 90a], which gives the added benefit that the presence of non-deterministic results is a sure sign of an error. Many errors in both TWOS itself and its simulations have been discovered through non-deterministic results. Some of these errors have been related purely to deterministic concerns, such as the method of ordering messages. Others, however, have been fundamental errors like losing messages, failing to roll back properly, or improper scheduling. The failure of TWOS' determinism brought these errors to light much more quickly than if we had not demanded deterministic results from the mechanism.

Some of the tools used in debugging TWOS itself rely on determinism, such as the event log tool discussed in section 4.5. These tools work on the assumption that the committed results of one correct run of the simulation much match those of another. By comparing certain portions of the results of two simulation runs that should match, but do not, problems can often be pinpointed.

Perhaps the greatest debugging benefit of TWOS' commitment to determinism for application debugging is that any error occurring in a simulation will continue to occur on every run. As a result, users can be certain that errors will not suddenly pop up, only to disappear when the run is repeated to try to isolate the problem. An error in user code will persist across all TWOS runs.

Also worth noting is that the guarantee of determinism implies that user code need never be concerned with issues of timing. Despite any timing variability, TWOS must produce deterministic results, therefore users need not worry about timing. If timing considerations actually cause non-determinism, that is a bug in TWOS and must be corrected.

Of course, TWOS only provides deterministic results for the simulation running on top of the system. TWOS itself does not run deterministically. Therefore, determinism-based tools cannot be used for many debugging problems under TWOS, and problems in TWOS itself may not recur when the system is rerun. In essence, TWOS takes on itself the burden of converting an inherently non-deterministic system into a deterministic one.

Non-deterministic results can signal a problem in TWOS itself. If the user has followed certain rules (which are required because the TWOS implementation does not trap all illegal user actions), his simulation should produce deterministic results. So, if those rules are followed, non-determinism signals an error in TWOS. Unfortunately, such errors usually will not recur deterministically, but at least the user has an indication that the error is present, and perhaps some clues about its source.

#### 4. Debugging Methods For TWOS

The first parallel version of TWOS was developed on experimental hardware, the Caltech/JPL Mark II Hypercube. Because this hardware was so new, the associated software was not yet mature. In particular, the debugging facilities were primitive. So, from the very first, TWOS needed to deal with debugging problems without much assistance from existing software. As the hardware platforms used for TWOS have matured, their debugging tools have improved, but they are still not sufficient. In some ways, the early lack of good debugging software proved helpful, as it required the development of custom debugging software specific to the TWOS system, rather than relying on general purpose software that did not know anything about the ways TWOS operated.

##### 4.1 TWOS Statistics

One of the most important decisions made early in the TWOS project was to keep careful statistics on all operating system actions. Given that TWOS would roll back and discard work on a routine basis when operating correctly,

only by keeping very careful track of what the system was doing could we hope to determine if it was operating correctly. Therefore, TWOS was designed to tabulate all actions it took. In particular, redundant statistics were chosen to allow independent crosschecks of correctness.

As a simple example, TWOS counts all messages sent by objects, and all messages received by objects. Since messages are not permitted to be cancelled in transit, any message sent must be received, so these two statistics must be equal at the end of a run. Similarly, separate counts are kept of the number of committed messages sent and received. These counts are different than the simple counts of messages sent and received, since TWOS cancels some messages. Again, the count of committed messages sent must match the count of committed messages received or an error has occurred. Perhaps a message cancellation failed, for instance. TWOS keeps many other redundant statistics for these purposes.

The statistics can be used to keep track of more complicated interactions. TWOS cancels messages by sending negative copies of those messages. When a negative and positive copy of the same message are in the same queue, they annihilate. TWOS keeps count of negative messages sent and received separately from positive messages. Every message sent must either be committed or cancelled. Therefore, subtracting the number of messages cancelled from the number of messages sent should yield the number of messages committed.

After completion of a run, all of these statistics are written out on a per-object basis. A tool called `check` is then used to make sure that all statistics balance correctly. If they do not, the failed balances are brought to the user's attention. This process has been of great value in detecting errors that do not cause crashes in the TWOS code. Often, a bug will give no visible signs of occurrence, except that it will cause a cancellation to be missed, or an extra copy of a message to be sent. Even looking at the user-level results of the run might not uncover any error, as the user usually does not know what results his simulation is supposed to produce, and the error might not actually show up in his results even if he did know what to expect. However, some other application might fail due to the same problem. Without the availability of these TWOS statistics, such a problem might not be discovered.

The code that calculates statistics must be written very carefully. Since balancing these statistics validates the run, any error in counting statistics can falsely indicate a problem. More than once, what appeared to be an execution error actually has turned out to be an error in calculating statistics. Not only the TWOS statistics code itself must be handled carefully, but also the `check` program. If changes to TWOS change the balancing equations used to test correctness, the `check` program must also be changed. For instance, initially states were only produced by events, so the number of committed states and



the number of committed events would always balance. However, once dynamic creation of objects was permitted, that action also produced a state, so the number of committed dynamic creations had to be added to the number of committed events to balance with committed states.

The overhead of gathering these statistics is quite low, compared to what each statistic counts. In general, adding to a statistic takes a few assembly language instructions, while the action being counted might take milliseconds. The amount of storage consumed by statistics gathering is also relatively modest, totalling perhaps 75,000 bytes in a typical run. Given that such a run will normally consume at least 3-4Mbytes, the statistics are not a major component in the storage requirements.

These statistics have uncovered many bugs. For example, TWOS contains a defined data type called "Int", not necessarily the same as the standard C data type "int". A variable that should have been declared as an "int" was changed to an "Int", resulting in its length being changed from 32 bits to 16 bits on one of the machines being used at the time. But the variable was being used to store the return value of a function that returned a "Long" (32 bits). Only rarely did this bug cause problems in the applications we used for testing. The only cases where the bug showed up resulted in sending duplicate messages for initialization purposes, resulting only in a single object's state being identically initialized twice. The user would have seen no difference in his program's results, but the count of committed messages was off by two, detecting this error before it had actually corrupted a user's program.

## 4.2 The TWOS Tester

TWOS contains a facility called the `tester` that is essentially an internal debugger. When certain errors occur, TWOS traps to the `tester`. While in the `tester`, the programmer can look at a wide variety of internal data structures. For instance, the programmer can print the scheduler queue on each node; the input, output, and state queues for each object; the object control block for any object; structures related to the locations of objects; the virtual times each node is operating at; and many other interesting pieces of information.

In essence, the `tester` gives some of the same functionality of a normal debugger. But it is both more and less than a normal debugger. The `tester` does not have the capability to print the contents of any variable in the TWOS code, for instance, and it has only limited breakpointing facilities. On the other hand, unlike a standard debugger, it understands TWOS data structures. The `tester` knows what a scheduler queue looks like, so it need not print out one entry at a time, forcing the programmer to follow pointers. It knows which fields of a data structure are likely to be needed and which need not be shown. It can use some of the TWOS facilities for locating objects to help a user find the node hosting a particular object.



The `tester` is the primary debugging tool for operating system problems in TWOS. It provides access to most of the information needed in determining what has gone wrong in a run.

Originally, the `tester` was used even more extensively. It was first set up to serve as a tool for interactively testing the correctness of various functions in TWOS. A new function that had just been written could be called directly by the `tester` with any parameter values the programmer wanted. Just because a piece of code worked well for the normal test cases that actual simulations used did not necessarily mean that it would work for extreme values or unusual combinations of values. Since forcing TWOS to produce such unlikely, but legal, sets of parameters directly was often very difficult, the `tester` provided a good means for quickly checking the correctness of a piece of code. This approach proved most valuable during the process of writing some of the most basic TWOS code, particularly before the system was in a state where it could work at all, as a whole. [Elshoff 88] describes a similar method used in debugging the Amoeba system.

As an example of the use of the `tester`, sometimes a TWOS application will get stuck, failing to make progress when it should, due to some error. This behavior can arise for a variety of reasons, including scheduler bugs, an infinite loop of messages sent for the current simulation time, memory exhaustion, and many others. The program can be stopped, the `tester` used to examine the scheduler queue, and the program restarted. Then the scheduler queue and message queues could be examined again, giving clues about the behavior of the application.

### 4.3 The Monitor

In some cases, determining the flow of control of TWOS is more useful than examining the results of a run with the `tester`. The TWOS monitor is used for such cases. The `monitor` is not normally built into TWOS, as it is somewhat clumsy to use (owing to hardware limitations early in the project) and slows the system down. TWOS must be recompiled to include the `monitor`. On the other hand, the `monitor` has a lot of flexibility once it has been set up. The `monitor` is primarily used for debugging TWOS itself.

The `monitor` allows the programmer to print a message every time certain routines are called. Routines can be flagged from a file read at run time, or interactively with the `tester`. The message indicates which routine has been called and the values of its parameters. If desired, the `monitor` can trap to the `tester` when a certain routine is called, allowing the programmer to instantly examine the state of the system before any of the routine's code is executed.

The `monitor` is generally used for particularly thorny problems, where the `tester` alone proves insufficient. A good general purpose debugger might

well provide the same capabilities. However, the monitor does correctly handle running on multiple nodes.

The monitor proved useful on several occasions. For instance, the monitor proved fast enough to catch some race conditions. When the message passing system on an early piece of hardware used by TWOS (the Mark II Hypercube) was being debugged, the TWOS monitor showed that there were race conditions in message broadcasting. Sometimes a race condition could cause a broadcast to send too few messages. A few days later, the monitor caught the less common case of a broadcast sending too many messages.

#### 4.4 The TWOS Progress Chart

Many of the worst TWOS problems have not had to do with correct operation, but with efficient operation. The goal of TWOS is to run simulations quickly, so, even if the results are correct, the system is fatally flawed if it doesn't achieve good performance. Performance problems are often very hard to diagnose in TWOS. Unlike correctness problems, one generally cannot quickly pin down the problem to one section of the code. Debugging performance problems is a common theme in parallel processing [Segall 85], [Socha 88].

Several TWOS tools are specifically designed to help with performance problems [Bellenot 89]. One of these is the progress chart. The progress chart is a graphical tool that plots lines on a screen for every event run during a simulation's execution, both committed and uncommitted. In a single picture, the progress chart can summarize the entire course of a TWOS run.

The progress chart works by keeping a detailed log of all event executions in TWOS. In principle, the person debugging TWOS could look at this log for insight into performance problems. However, a typical TWOS run might have over 300,000 committed events, and perhaps half as many more events that were rolled back. The event logs for such simulations are far too large to scan manually.

The progress chart uses this data to plot a graphical display in which each event execution is represented by a line on the display. The display plots real time versus simulation time, as shown in figure 1. In most cases, the lines are so short that they appear as points in the normal display.

This chart shows several interesting features. First, it gives an idea of the progress of the simulation. Areas of the chart that are fairly flat suggest that little progress is being made in simulation time for a long period of real time. Either the simulation has a lot of work to do in that span of simulation time, or TWOS is not doing a very good job of speeding things up during that span. Areas of the chart with large slopes indicate that TWOS is speeding over those spans of simulation time.

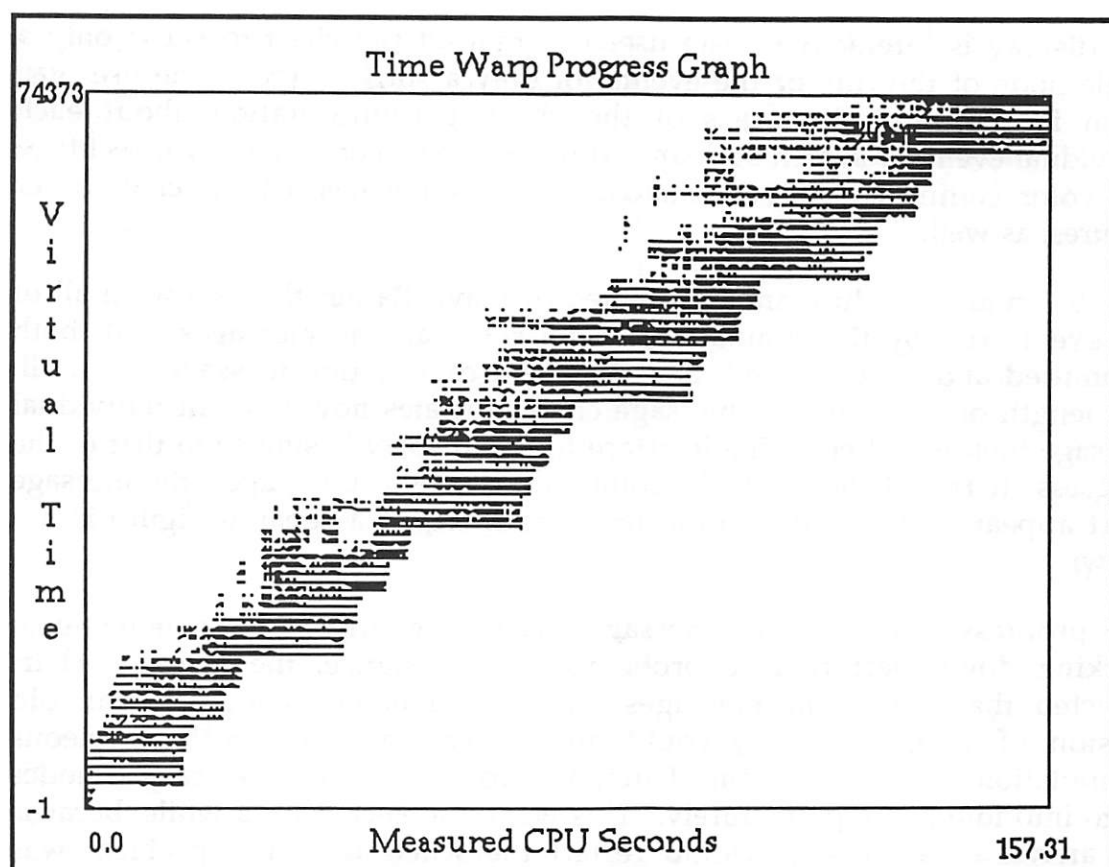


Figure 1: A Time Warp Progress Chart

The chart also gives an idea of the range of simulation times being executed on the various nodes at the same real time. A broad vertical spread indicates that some nodes are very far ahead of others at that point. A narrow spread indicates that all nodes are working within a small band of simulation time at that point.

The chart shows another interesting feature of the simulation, the progress of global virtual time in real time. The lower bound of the graph is the virtual time at which the earliest event is occurring at any real time. This bound is precisely GVT. Thus, examining this chart can show which parts of the simulation had quick GVT progress, and which parts progressed slowly.

The progress chart is a color display whose colors can be used in two ways. First, the chart's lines can be color coded to the types of object in the simulation. For instance, in the colliding pucks simulation whose chart is shown in figure 1, pucks' events are colored blue, while sectors' events are colored red. The simulation designer can rather easily set his own color scheme. Alternately, colors can be used to highlight which events are

committed and which rolled back. This color coding can help determine the rollback behavior of the simulation.

The display is interactive. The user can request the chart to show only a single node of the run, or the events for only a single object. The user can zoom in on particular areas of the chart, get information about each individual event, or switch back and forth between color coding by object type and color coding by rollback status. The display has a number of other features, as well.

TWOS can also produce another related display. Rather than showing all of the events run by the simulation, it can show all the messages sent, both committed and uncommitted, and the event cancellation messages, as well. The length of a line on the message chart indicates how long an individual message took to deliver. The interface to this display is similar to that of the progress chart. At the level of resolution possible in this paper, the message chart appears very similar to the progress chart, so a separate figure is not shown.

The progress chart and the message chart have proven very valuable in tracking down performance problems. For instance, the message chart detected that TWOS antimeessages did not travel fast enough in an old version of TWOS, so they could not always catch up with erroneous computations and cancel them. Later, an error in the scheduler caused nodes to go into idle mode prematurely. This went undetected for a while, because the arrival of a message would restart the scheduler. The problem was noticed when a simple test simulation with very few messages took too long to run. Occasional system messages would turn the schedulers back on for a short time, but then the system would go idle until the next round of system messages. The message plot showed these broad gaps of inactivity. More recently, the message plot has shown the negative effects of paging on the Mach version of TWOS.

[Lehr 89] describes a graphical tool bearing some resemblance to the progress chart. However, it does not include a concept of virtual time, so it does not show progress in virtual time versus real time.

One improvement necessary for both the progress chart and the message chart is to permit selective tracing of particular objects or periods of time during the run. The logs necessary to run these tools tend to be very large and the memory requirements for storing the data can sometimes prove burdensome. A similar filtering approach is used by [Elshoff 88], and many others. This improvement will be made when time permits.



## 4.5 The Event Log

TWOS can keep a log of committed events for a simulation. This log has one entry for each event, describing the object performing the event, the simulation time of the event, and the object sending the message that caused this event. If a simulation is producing non-deterministic results, indicating an error in TWOS, the event log can be used to track down the problem. Logging and replay is a commonly used method in parallel debugging [Lin 88], [Cheung 90], though TWOS' use of the method has certain wrinkles not present in other systems.

The event log is used in two ways. First, the sequential simulator can also produce an event log. That log can be compared against the event log for an incorrect TWOS run to find the point of divergence. Knowing precisely the first event that produced different results from the correct run can be very helpful in tracking down the problem.

Sometimes, though, knowing where divergence occurred is not, itself, enough. If the problem is internal to TWOS, the event log may not contain enough information to determine the cause of the error. In such cases, the event log can be used in another way. TWOS can read a correct event log into memory and start a run. At the point of first divergence of committed results for this run from the event log known to be correct, TWOS will print an error message and call the tester, allowing the debugger to thoroughly investigate the state of the machine.

The event log can also be useful for certain types of simulation debugging. For instance, a simulation programmer might have replaced sorting algorithms with more efficient ones, still expecting to get the same sorted results. If an error in the new version causes the simulation to produce improper results, an event log from the old version can be compared to the new version's event log to track down where the error first occurs.

Recently, the event log was used to detect differences in the floating point algorithms used by two different machines. Both machines used Motorola 68020 processors and compilers from the same manufacturer, but a simulation ran differently on the two machines. By taking event logs of the two runs of the simulation, comparisons were made that pinpointed the first event at which divergence occurred, which led to detection of differences in the floating point arithmetic algorithms present in both the hardware and the software.

## 4.6 Paranoid Code

TWOS' goal is speed, so the system tries to avoid unnecessary tests and checks that would slow down normal execution. When an error occurs, however, it often could have been trapped before it caused a crash or otherwise destroyed



the information necessary to find it if the system had contained code to check for potential problems. TWOS contains a lot of code of this sort, but it is "paranoid" code – it is normally not compiled into the system. When a new capability is being added to TWOS, or a problem has arisen, the system is recompiled with the paranoid code included. Frequently, this code will instantly spot a problem.

Paranoid code typically consists of tests performed on every one of some common operation. For instance, TWOS contains many lists, so it has generic code for creating list elements, deallocating list elements, linking and unlinking them, and so forth. These operations are performed millions of times during a typical TWOS run, so they are implemented in a very simple, efficient way. However, sometimes system errors arise that corrupt list element headers, or fail to unlink them before deallocating them, or otherwise do not follow the rules of handling list elements. TWOS contains paranoid code that tests the validity of list element headers every time they are operated on, to ensure that the operation is valid at that point, and that the headers haven't been corrupted.

One of the first actions taken when a version of TWOS begins to crash is to recompile it with the paranoid code enabled. Frequently, the paranoid code will trap the error before it gets too far the next time it occurs.

#### 4.7 Special Purpose Tools

The TWOS project has used several special purpose tools to track down problems in particular parts of the code. One such tool is called the Hypercircle. When TWOS was running on the Caltech/Mark3 hypercubes, certain performance problems occurred that might have been caused by communications bottlenecks between the nodes, or by differences in the time necessary to travel between nearby nodes versus far away nodes. The Hypercircle was designed to test this hypothesis.

The Hypercircle consisted of a graphics display and associated text. The graphics display drew a 32 node hypercube architecture in three dimensions. Initially, each pair of nodes having a physical connection was connected by a faint line on the display. As the Hypercircle program ran, reading in a record of a TWOS run, every message sent in the TWOS run was represented on the display by a temporary bright line traversing the path from source node to destination node actually taken in the hypercube. Negative messages were shown in a different color than normal messages.

Each arc of the Hypercircle grew brighter and thicker as messages travelled along it. As time went on, any arcs that were particularly heavily used (or lightly used) would begin to stand out noticeably in the pattern. Simultaneously, a running display at the bottom of the screen showed the

elapsed time necessary to send messages over a different number of hops, from 1 to 5.

This display clearly, graphically demonstrated that network contention was not the problem, nor were inordinate delays in messages travelling over long paths versus those travelling over short paths. The performance problems proved to have more to do with internal handling of messages at the source and destination nodes than with any delays in getting them from one to the other.

Another tool has helped in debugging TWOS' dynamic load management facility [Reiher 90b]. The load manager is supposed to move load from "heavily loaded" nodes to "lightly loaded" nodes, where "load" is a quantity rather specific to TWOS and its optimistic method of execution. The same basic logging code used to produce the event log was quickly adapted to produce a log of loads on different nodes at different points in the simulation. The log also holds information about migrations performed to implement load management policies. This information was then used to determine the correctness and efficacy of the load management policy.

This log has also been used to feed a graphical dynamic load management display. This display can run in either single step or continuous mode, and clearly shows load being transferred from heavily loaded nodes to lightly loaded nodes, and the subsequent evening of the loads on the two nodes. It also shows how much information had to be transferred to accomplish a migration, giving some idea of the cost of the migration.

Another method used to debug TWOS is to write test applications that stress particular aspects of the system. Certain applications meant to induce cascading rollbacks have uncovered problems with the message delivery system and the handling of system messages. One such application, described in [Bellenot 89], was called "sloooow". This "target and arrows" simulation had lots of fast arrow objects that shot messages at a rather slow target object. The node hosting the target object would run out of memory much faster than the nodes hosting arrow objects, uncovering flow control problems.

Invariably, new features added to TWOS received their most serious debugging only when an actual application started to make extensive use of them. To some extent, test simulations would uncover certain problems in the features, but only actual patterns of usage would uncover the full range of flaws in the methods. The TWOS project has been fortunate enough to have an associated simulation development project that has provided realistic, complex benchmarks that have been of tremendous value in finding bugs and fixing performance problems.

## 5. Conclusions

TWOS has been a challenging system to debug. First, it is an operating system that has close control of its hardware, thus giving it many opportunities to make disastrous choices. Second, it runs on parallel or distributed hardware, adding many possibilities for errors based on asynchrony and timing. Third, it uses an unusual synchronization method that was unproven at the start of the project, and that still tends to defy intuition. Fourth, much of the hardware used for development had only primitive debugging software available. Fifth, it is a research system devoted to working with fairly risky methods, so no existing algorithms could be adapted for many important system functions.

Our experience in debugging TWOS should be instructive to others developing complex parallel and distributed systems. The value of deterministic results in debugging was great. The TWOS experience with providing deterministic results suggests that other distributed systems projects should consider attempting to provide determinism at the user level, for debugging reasons, if for no other purpose, even if the synchronization method is not synchronous.

TWOS' extensive use of redundant statistics for error detection has proved invaluable, and is a technique that should be used by all system developers. We have long since lost track of how many bugs were discovered only due to problems in the statistics. We regard the early decision to keep redundant statistics for crosschecking to be the single best decision made in the course of this project.

One important lesson learned from the TWOS debugging effort is that any ambitious systems project must budget time for the development of debugging tools. The tools will have to be developed, one way or another, and watching for opportunities to develop them will save time in the long run. Not recognizing this fact early in the project was quite expensive for TWOS. Once personnel with an understanding of the importance of strong debugging support arrived, progress became much more rapid.

The value of interactive graphical tools, particularly the progress chart and the message chart, was great. These tools allowed developers to pinpoint many subtle problems by giving an overall view of the system's behavior, while simultaneously allowing more detailed examination of suspicious areas of the charts.

The TWOS experience also points up the importance of testing a system with pathological application programs. While the system should certainly not be tuned to handle unlikely situations at the cost of normal ones, understanding how the system behaves in extreme cases is vital.

Most of the TWOS debugging tools are not revolutionary. Some of them are familiar to most software engineers, and some of them, while new, are so specific to the TWOS problem that they are unlikely to be of direct use to many other groups. However, the overall approach TWOS takes to debugging provides an interesting case study of successfully applying existing techniques and inventing new techniques to ease in the debugging of an experimental research distributed system.

### References

- [Bellenot 89] S. Bellenot and M. Di Loreto, "Tools For Measuring the Performance and Diagnosing the Behavior of Distributed Simulations Using Time Warp," *Proceedings of the 1989 SCS Conference on Distributed Simulation*, Vol. 21, No. 1, 1989.
- [Cheung 90] W. Cheung, J. Black, and E. Manning, "A Framework For Distributed Debugging," *IEEE Software*, Jan. 90.
- [Elshoff 88] I. Elshoff, "A Distributed Debugger For Amoeba," *ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, May 1988.
- [Emrath 88] P. Emrath, D. Padua, "Automatic Detection of Non-Determinacy in Parallel Programs," *ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, May 1988.
- [Fujimoto 90] R. Fujimoto, "Parallel Discrete Event Simulation," *Communications of the ACM*, vol 33., no. 10, Oct. 90.
- [Hontalas 89] P. Hontalas, B. Beckman, M. Di Loreto, L. Blume, P. Reiher, K. Sturdevant, L. V. Warren, J. Wedel, F. Wieland, and D. Jefferson, "Performance of the Colliding Pucks Simulation on the Time Warp Operating System (Part 1: Asynchronous Behavior and Sectoring)," In *Proceedings of the SCS Multiconference on Distributed Simulation*, Unger, B. and Fujimoto, R., Eds., Society For Computer Simulation, San Diego, CA, 1989.
- [Jefferson 85] D. Jefferson,, "Virtual Time," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 3, 1985.
- [Jefferson 87] D. Jefferson, B. Beckman, F. Wieland, L. Blume, M. Di Loreto, P. Hontalas, P. Laroche, K. Sturdevant, J. Tupman, V. Warren, J. Wedel, H. Younger, and S. Bellenot, "Distributed Simulation and the Time Warp Operating System," *Proceedings of the 11th Symposium on Operating System Principles*,, 1987.

- [Lehr 89] T. Lehr, Z. Segall, D. Vrsalovic, E. Caplan, A. Chung, and C. Fineman, "Visualizing Performance Debugging," *Computer*, vol. 22, no. 10, Oct. 1989.
- [Lin 88] C. Lin and R. LeBlanc, "Event-Based Debugging of Object/Action Programs," *ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, May 1988.
- [Reiher 90a] P. Reiher, F. Wieland, and P. Hontalas, "Providing Determinism In the Time Warp Operating System – Costs, Benefits, and Implications," In *Proceedings of the IEEE Workshop on Experimental Distributed Systems*, October 1990.
- [Reiher 90b] P. Reiher and D. Jefferson, "Virtual Time Based Dynamic Load Management In the Time Warp Operating System," *Transactions of the Society for Computer Simulation*, vol.7, no. 2, July 1990.
- [Segall 85] Z. Segall and L. Rudolph, "PIE: A Programming and Instrumentation Environment for Parallel Processing," *IEEE Software*, vol. 2, no. 6, Nov. 1985.
- [Socha 88] D. Socha, M. Bailey, and D. Notkin, "Voyeur: Graphical Views of Parallel Programs," *ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, May 1988.



## Lock Granularity Tuning Mechanisms in SVR4/MP

Mark Campbell, [mark.campbell@ncrcae.Columbia.NCR.COM](mailto:mark.campbell@ncrcae.Columbia.NCR.COM)  
Russ Holt, [russ.holt@ncrcae.Columbia.NCR.COM](mailto:russ.holt@ncrcae.Columbia.NCR.COM)  
John Slice, [john.slice@ncrcae.Columbia.NCR.COM](mailto:john.slice@ncrcae.Columbia.NCR.COM)

NCR Corporation -- E&M Columbia  
3325 Platt Springs Road  
West Columbia, South Carolina 29169

### Abstract

The tuning of lock granularity in multiprocessing operating systems has tended to be rather ad hoc in the past; in this paper we present the tools developed for the multi-threading of SVR4 (UNIX System V Release 4). The resulting operating system, SVR4/MP, supports symmetric tightly-coupled multiprocessing and has been implemented on both an MC88000-based multiprocessor and an I80386-based multiprocessor. The lock granularity tuning tools used in the development of SVR4/MP are designed to facilitate the rapid, iterative tuning of lock granularity of kernel subsystems; therefore they are an excellent tool set for developing drivers, file systems, and other add-on kernel features.

### 1. Introduction

Both the number and placement of locks to insure mutual exclusion of sections of kernel code in an operating system supporting tightly-coupled multiprocessing (OS/MP) are major factors in the performance and scalability of that operating system. There is in general an inverse relationship between the number of locks used and the scope of each lock -- as the number of locks increase the scope of each lock (i.e., the number of resources protected by that lock and/or the duration of the protection of that resource) decreases and as the number of locks decrease the scope of each lock increases. If the designers of an operating system use too few locks, then the system will not in general scale well as the number of processors increases; if too many locks are implemented, processor cycles will be wasted in lock acquisition and subsequent lock deacquisition. The term *lock granularity* is generally used to describe the number of locks and scope of each lock used in a system.

OS/MP's can be divided roughly into three categories with respect to lock granularity: coarse-grained, medium-grained, and fine-grained. The end-case of a coarse-grained OS/MP is one which supports a single lock protecting access to all shared OS/MP components; the end-case of a fine-grained OS/MP is one which uses a unique lock for each shared OS/MP component (e.g., each element in an OS/MP shared structure being protected by a lock dedicated to that element). The most common taxonomy of OS/MP's tends to define a coarse-grained OS/MP as one which supports from one to two processors in a relatively inefficient manner (e.g., the 1-lock Purdue VAX kernel), a medium-grained OS/MP as one which supports from one to four processors in a relatively efficient manner (e.g., the 25-lock SCO MPX kernel), and a fine-grained OS/MP as one which supports more than four processors efficiently (e.g., the 128-lock SVR4/MP kernel).<sup>1</sup> There are varying degrees of

1. This is a religious issue much akin to benchmarking -- some would say that the SCO MPX kernel scales perfectly on many commercial benchmarks (e.g., multiple instances of the Dhrystone benchmark). This rough taxonomy relies on the use of roughly equivalent application loads which cause a non-trivial amount of contention for shared OS/MP resources.

lock granularity even within these somewhat heuristic categories; thus a major factor in constructing an OS/MP concerns the granularity of the locking.

There are three basic problems then associated with OS/MP lock granularity:

- How does a developer determine the initial proper locking granularity in an OS/MP?
- How does a developer determine the final proper locking granularity in an OS/MP?
- How do subsequent developers determine the proper locking granularity in other OS/MP components (e.g., updates, drivers, file systems, etc.)?

We believe that too much attention has been focused on determining the initial locking granularity of an OS/MP and not enough on rapid, iterative tuning of the placement and granularity of locks in initial OS/MP implementations. In other words, in most OS/MP implementations too much emphasis is placed on qualitatively determining lock placement through a set of heuristics comprised of measurement performed on uniprocessing operating systems and "common sense". In most developments, we have found that two developers with years of multiprocessing experience tend to disagree on relatively straightforward lock placement issues. We also believe that there is a dearth of existing tools in OS/MP implementations which allow subsequent OS/MP components be multi-threaded properly.

In other OS/MP implementations an iterative approach based on subsystem locks has been developed by which subsystems are multi-threaded one at a time and integrated into an otherwise single-threaded kernel. We rejected this approach because even a single major UNIX subsystem (e.g., process management, STREAMS, VM, VFS, a VFS file system, etc.) radically affects the overall performance and scalability of an OS/MP. The major reason for this is the incredible amount of coupling between subsystems in UNIX.

SVR4/MP is a fine-grained OS/MP based on SVR4 (UNIX System V Release 4) which was designed to scale efficiently from one to thirty-two processors. The initial placement and granularity of locks within SVR4/MP was de-emphasized; most of the development effort was focused on rapid, iterative tuning of lock granularity. The placement and granularity of locks in SVR4/MP were driven by a set of sophisticated tools which were used to perform iterative overall performance/scalability tuning of the operating system. The tools which were developed to support the quantitative determination of locking granularity in SVR4/MP are described in this document. An example of the use of these tools to optimize overall system efficiency by increasing and decreasing the lock granularity of SVR4/MP is also included.

## 2. Lock Information

The strategy for the multithreading of the SVR4 required a provision for an evolution path to allow for performance tuning. Although a primary goal was to have a multithreaded kernel available in a short time span, a poorly performing kernel was clearly unacceptable. Optimizing code that seldom gets executed shows little improvement, and predicting which areas of the kernel need optimization based on qualitative analysis is a low percentage gamble. Interacting processes running on a multiple processor system do not react predictably to tuning attempts. The reduction or elimination of a major bottleneck may reduce or eliminate others. Or, new ones may appear that mitigate the benefit of the optimization. The profiling capabilities of SVR4/MP were designed into the locking primitives from the beginning so that the tuning effort could be directed at areas that would benefit most from optimization.

Some multiprocessing performance issues to be monitored included latency, granularity and scope, and frequency and contention. Latency is the delay involved in waiting for a locking primitive to become available. This delay depends on the work performed inside the critical region and how frequently the region is entered. Granularity and scope are measures of the size of a resource. This size refers not only to the amount of data associated with a resource but also involves the actions

applicable to that data. Granularity may be coarse-grained as in a global lock for a set of associated data; or, the granularity may be fine-grained as in a lock per data entity. Resources which have high contention rates should have a higher degree of granularity. The scope refers to the number of instances of the lock (i.e., locations in which the lock is placed). Frequency and contention are measures of how often a resource is requested. If a resource is not frequently accessed it might be adequately controlled by a broad scope lock.

In order to measure these performance issues properly, statistics must be maintained for the locking primitives. The locking primitives used for SVR4/MP allowed for an attachment of a lock information structure. The lock information structure was used for tracking design and performance statistics.

## 2.1 Lock Information Structure

The lock information structure is provided as an integral part of the locking implementation for SVR4/MP. The *LockInfo* structure is shown in figure 1.

```
struct LockInfo {
    unsigned long    mask;
    unsigned long    rank;
    CommonLock_p    lock;
    LockInfo_p      list;
    unsigned long    get_hits;
    unsigned long    get_misses;
    unsigned long    get_spins;
    unsigned long    get_higher;
    unsigned long    get_lower;
    unsigned long    get_sleeps;
    unsigned long    try_hits;
    unsigned long    try_misses;
    unsigned long    spins_on_restore;
    unsigned long    bad_sleep;
    unsigned long    attaches;
    unsigned char    mask_flag;
    unsigned char    going_to_sleep;
    unsigned char    going_to_wakeup;
    unsigned char    restored;
};
```

Figure 1. The Lock Information Structure

If the lock information feature of the locking primitives is enabled, then a lock information structure is attached to the lock to provide for the tracking of lock statistics. The design of the *LockInfo* structure provides two modes of attachment: *PerType* and *PerLock*. The *PerType* mode enforces the sharing of the statistics for all instances of a lock. The *PerLock* mode allows for the association of a lock information structure for each instance of the lock. The current implementation supports only the *PerType* mode.

The *LockInfo* structure is used to hold information for design verification and for collecting information on lock utilization. The design validation information is available to check the placement of locks and to detect potential deadlocks. For performance tuning, we are concerned primarily with the utilization information. We will address specifically the use of the following statistics:

- *get\_hits*  
This statistic tracks the number of successful attempts to acquire the lock.

- *get\_misses*  
This statistic tracks the number of unsuccessful attempts to acquire the lock.
- *get\_spins*  
This statistic tracks the number of simple spin loops necessary before the lock became available after a miss.
- *spins\_on\_restore*  
The *spins\_on\_restore* field in the *LockInfo* structure was originally called *thundering\_herd*. The thundering herd scenario specifically refers to the waking up of many processes of which only one will obtain the resource. The unfortunate processes must go back to sleep. The *spins\_on\_restore* field may be used as an indicator of the thundering herd scenario. The field is actually a lock which was restored and released again even though no free of the lock had occurred. This field may also be an indicator of a coarse-grained lock. The thundering herd scenario would be indicated by a high *spins\_on\_restore* count for locks around the sleep/wakeup paradigm with uniprocessor style locks (flags with busy and wanted indications).

## 2.2 Lock Information Driver

The *LockInfo* driver is a pseudo-driver provided for accessing the statistics information. The *LockInfo* Driver can be configured into the SVR4/MP kernel by including the module *LockInfo* in the configuration systems file. The *open*, *read*, and *close* functions provide the primary interface. For some of the information devices a reset function is provided via an *open* for write mode. The reset clears the data associated with that information device. This function is especially useful when snapshots are taken. The following portrays a method for taking a contention snapshot of the Oracle TP1 benchmark:

```
> /dev/Lock/Contention
RunOracleTP1
cat /dev/Lock/Contention > /tmp/Results
```

The *LockInfo* device may be accessed by making the appropriate device nodes. The current devices available for performance tuning are *SpinsOnRestore*, and *Contention*. A suggested */dev* layout follows:

```
/dev/Lock:
crw----- 1 root   rootgrp  77, 2 ... SpinsOnRestore
crw----- 1 root   rootgrp  77, 3 ... Contention
```

### 2.2.1 SpinsOnRestore Device

A *cat* of the *SpinsOnRestore* device displays the number of instances in which a lock is restored and then released again without a free of the lock occurring. This may serve as an indicator of a coarse-grained lock or the thundering herd scenario. The format of the list is:

```
_LockName  NumberOfSpinsOnRestore
```

```

.
.
.
```

The *SpinsOnRestore* device supports the reset function.

### 2.2.2 Contention Device

A *cat* of the *Contention* device displays the number of attaches to a lock, and the number of hits, misses, and spins on the lock. The format of the list is:

<u>LockName</u>	Number of Attaches	Hits	Misses	Spins
	.			
	.			
	.			

The number of attaches is the number of locks associated with a particular *LockInfo* structure. For instance, if locks are placed within a structure, the number of structures determines the potential number of attaches<sup>2</sup> -- potential due to the dynamic determination of the attachment. Once attached, the attachment remains until the lock structure itself is freed or cleared.

The *Contention* device supports the reset function. The hits, misses, and spins are cleared.

### 3. Utilizing Lock Information For Tuning

The lock information can be used in profiling the lock implementation for performance tuning. The *get\_hits* field indicates the number of successful attempts to acquire the lock. The ratio of *get\_misses/get\_hits* is a measure of contention. If the ratio is high, a more narrow scoped lock should be considered. Narrowing the scope of a lock means releasing and reacquiring the lock more frequently to reduce contention and latency on the lock. If the ratio is low, a more broad scoped lock may be necessary. Broadening the scope of a lock means holding it longer to reduce the number of times it needs to be acquired and released. The ratio *get\_spins/get\_misses* is also a measure of contention. A high ratio may indicate a need to sleep rather than spin when attempting to acquire a lock. It may also indicate the lock is too coarse-grained. The ratio of *get\_spins/get\_hits* is an indication of latency on a particular lock. The next section describes one example of using the *Contention* device to reduce and eliminate some locking problem areas.

#### 3.1 An Example Using The *Contention* Device

The following describes how the *Contention* device was utilized in tuning SVR4/MP for the Oracle TP1 benchmark. A script was used to convert the raw contention data into a format for study.

2. A *LockInfo* structure associated with all instances of the lock is defined as *PerType*. A *PerLock LockInfo* structure would be implemented via an allocation of a *LockInfo* structure for each structure containing the lock. The *PerLock* version of *LockInfo* is not yet supported.



LockInfo Structure	Attaches	Hits	Misses	Spins	S/M	S/M/A
_AddressSpaceLockInfo	1	5244	3	5947	1982.333	1982.333
_AnonymousMapLockInfo	1	1516	269	235027	873.706	873.706
_ClockLockInfo	1	25601	36	15973	443.694	443.694
_TimeSharingClassParmTableLockInfo	1	19418	981	188000	191.641	191.641
_ItimerLockInfo	1	14679	609	109958	180.555	180.555
_sd_SysCallSerializationLockInfo	10	113	23	38697	1682.478	168.248
_PermSyncLockInfo	3	39360	2322	920125	396.264	132.088
_PageTableLockInfo	1	9656	80	9571	119.638	119.638
_DispatcherQueueLockInfo	1	49979	382	30508	79.864	79.864
_SleepHashQueueLockInfo	1	25549	708	56192	79.367	79.367
_PageFreeListLockInfo	1	2356	29	2031	70.034	70.034
_AnonymousPageLockInfo	1	699	2	140	70.000	70.000
_SemLockInfo	8	17671	343	179363	522.924	65.366
_SwapLockInfo	1	2464	8	504	63.000	63.000
_mdbfreelist_LockInfo	1	324	4	193	48.250	48.250
_PagesLockInfo	1	353	3	128	42.667	42.667
_TSTTY_LockInfo	1	1554	8	249	31.125	31.125
_AnonymousInformationLockInfo	1	823	2	49	24.500	24.500
.						
_bhdrlist_LockInfo	1	23	0	0	0.000	0.000
_bfreelist_LockInfo	1	8471	0	0	0.000	0.000
_basyn_LockInfo	1	30	0	0	0.000	0.000
.						
.						
.						

Figure 2. Contention Snapshot of Oracle (4 Generators Pass 1)

Figure 2 reveals potential lock contention problem areas. The *AddressSpaceLock* was researched and modified to not include user address spaces (since user address spaces are unique per process). This modification reduced the hits and eliminated the misses on this lock.<sup>3</sup>

The *AnonymousMapLock* was originally implemented as a global lock. As a global lock, the contention was observed as very high. By increasing the granularity, implementing as a per structure lock, the misses were reduced.

The results of these modifications are displayed in figure 3. The figure also reveals new areas to consider. Several new locks have bubbled to the top of the list. Of particular interest, the *PageTableLock* climbed from 80 misses to 900 and the spins increased tremendously.

3. The locking of user address spaces may be necessary for implementing *threads*. If so, the lock of the user address space may be applied as a lock per address space.

LockInfo Structure	Attaches	Hits	Misses	Spins	S/M	S/M/A
_bfreelist_LockInfo	1	4875	1	566	566.000	566.000
_ClockLockInfo	1	25062	60	22987	383.117	383.117
_AnonymousPageLockInfo	1	569	85	27930	328.588	328.588
_PagesLockInfo	1	328	7	2085	297.857	297.857
_PageTableLockInfo	1	9063	900	208208	231.342	231.342
_TimeSharingClassParmTableLockInfo	1	17982	1134	228912	201.862	201.862
_ItimerLockInfo	1	14487	658	124823	189.701	189.701
_sd_SysCallSerializationLockInfo	10	111	9	15122	1680.222	168.022
_ProcessFreeListLockInfo	1	664	1	165	165.000	165.000
_SwapLockInfo	1	2516	118	15564	131.898	131.898
_AnonymousInformationLockInfo	1	657	7	694	99.143	99.143
_PermSyncLockInfo	3	38709	1854	497376	268.272	89.424
_TSTTY_LockInfo	1	1680	9	796	88.444	88.444
_PageFreeListLockInfo	1	2138	34	2445	71.912	71.912
_SleepHashQueueLockInfo	1	23189	865	59028	68.240	68.240
_DispatcherQueueLockInfo	1	44815	422	21581	51.140	51.140
_mdbfreelist_LockInfo	1	517	3	132	44.000	44.000
_LdtermLockInfo	4	287	2	309	154.500	38.625
_PageListLockInfo	1	25701	231	8512	36.848	36.848
.	.	.	.	.	.	.
_Km_LockInfo	1	4278	3	91	30.333	30.333
_AnonymousMapLockInfo	39	1252	81	82569	1019.370	26.138
_STREAM_HeadLockInfo	10	148	3	681	227.000	22.700
.	.	.	.	.	.	.
_BhashLockInfo	59	2437	0	0	0.000	0.000
_AddressSpaceLockInfo	1	4801	0	0	0.000	0.000
_AcctVnodeLockInfo	1	7	0	0	0.000	0.000

Figure 3. Contention Snapshot of Oracle (Pass 2 Anonymous Map Redesign)

This information also may reveal locks with too narrow a scope (too many instances of the lock). This case was noted in the *PageListLock*. The traversal of the dirty list of pages was acquiring and releasing the lock for each page on the dirty list. As a result, the number of hits for this lock was high. The scope was enlarged to hold the lock for the duration of the traversal in some cases and the number of hits were reduced. The results of this modification are not shown, but was observed in improved benchmark performance.

#### 4. Conclusion

SVR4/MP met its dyadic scalability goals in less than nine months from its conception and within three weeks of its original schedule and was the first fully multithreaded fine-grained tightly-coupled multiprocessing implementation of SVR4. The strategy of quantitative lock placement was a major reason for the success of SVR4/MP -- without the tools discussed in this paper, quantitative lock placement and iterative scalability tuning would not have been possible.

We believe that the increasing standardization of kernel-level interfaces in SVR4 (e.g., DDI/DKI, VFS, etc.) coupled with the support of fine-grained tightly-coupled multiprocessing will make tools such as those discussed herein indispensable in the development of kernel-level code.

## 5. Acknowledgements

The first generation of NCR debug and tuning tools were developed by Todd Davis. These first generation debug and tuning tools formed the basis for those used in SVR4/MP.

## 6. References

1. Bach, M., Buroff, S., "Multiprocessor UNIX Operating Systems", *AT&T Bell Laboratories Technical Journal*, October 1984.
2. Boykin, Joseph, Langerman, Alan., "The Parallelization of Mach/4.3BSD: Design Philosophy and Performance Analysis", *Proceedings of the USENIX Workshop on Experiences with Distributed and Multiprocessor Systems*, October 1989.
3. Campbell, Mark, et. al., "The Parallelization of UNIX System V Release 4.0", *USENIX - Winter '91*, Dallas, TX.
4. Hamilton, G., Code, D., "An Experimental Symmetric Multiprocessor ULTRIX Kernel", *Proceedings of the Winter USENIX Conference*, 1988.
5. SCO, *SCO MPX -- A Technical Background Paper*, October 1989.
6. Sinkewicz, U., "A Strategy for SMP ULTRIX", *Summer 1988 USENIX Conference Proceedings*, 1988.
7. Smith, Bud E., Satchell, Stephen T, and Clifford, Heather B., "Multiprocessing Power on the Desktop", *Personal Workstation*, March 1990.

# Measured Performance of Caching in the Sprite Network File System

Brent B. Welch  
Xerox PARC  
3333 Coyote Hill Road  
Palo Alto CA 94304

welch@parc.xerox.com

## Abstract

This paper reports on the effectiveness of the caching strategy used in the Sprite network file system based on data taken over several months of day-to-day usage by a variety of users. Measurements include cache consistency activity, long term I/O traffic rates, long term cache hit rates, and the averages and variations in the size of the variable-sized caches. Network traffic is compared with traffic to the local cache, and the effects of paging traffic are considered. The overall conclusion is that the caching system is quite effective and poses a low overhead. Using a delayed write strategy, about half of the data written to client caches is never written through to a server, and less than 1% of the open operations by clients resulted in cache consistency actions by a server. †

January 29, 1991

## 1. Introduction

This paper presents performance measurements of the caching subsystem of the Sprite distributed file system [Ousterhout88]. In a Sprite network, client workstations are usually diskless, and the file servers implement different parts of a uniformly shared file system that provides the semantics of a 4.3 BSD UNIX timesharing system. Both clients and servers cache file data in their main memories to optimize I/O operations [Nelson88b]. Cache consistency is provided so that a read returns the most recently written data regardless of the way files are shared. A delayed write policy is used on the clients and servers, and it provides two benefits. First, applications do not have to wait for the relatively slow network and disk operations because writes occur in the background. Second, the delay period means that data written by applications can be deleted or overwritten without being written through to the file

---

† This work was done at the University of California, Berkeley, and was supported in part by the Defense Advanced Research Projects Agency under contract N00039-85-C-0269, in part by the National Science Foundation under grant ECS-8351961, and in part by General Motors Corporation.

servers.

The main results presented in this paper are summarized in Table 1. Less than 1% of the files opened triggered server consistency actions, which indicates that the consistency scheme poses low overhead. Concurrent read sharing is quite common, mainly due to executable files. The variable-sized caches adapt to the different needs of clients and servers, with servers using more memory for their file cache than clients. The read miss rate for client caches is good, but the low client write traffic ratio is even more significant. About half the data written by applications is never written through to the file servers. This data is deleted or overwritten within the 30-second delay period.

These results are based on statistics taken from our Sprite network from July through December, 1989, and a follow up study made in January 1991. At the time of the original study, the Sprite network was composed of four file servers (one Sun-4/280, 2 Sun-3s, and 1 DECstation 3100) and about 30 client workstations (11 Sun-3s, 4 Sun-4s, and 15 DECstation 3100s). One year later the network had around 36 clients (6 Sun-3s, 13 Sun-4s, and 17 DECstation 3100s), the two Sun-3 servers were retired and another Sun-4 server was added for experiments. Sprite is 4.3 BSD compatible, and was used for all the day-to-day computing needs of twenty or more grad students, a few professors, and a couple of staff members. There were about a dozen more other users that used the system occasionally. The results were obtained from raw data in the form of about 450 different statistics that were maintained in the kernel and periodically sampled. File servers were sampled hourly, and clients were sampled 5 times each day (at 8am, 11am, 2pm, 5pm, and 8pm). The data was recorded for a 6 month period. However, some new statistics were introduced late in the study period to focus on particular metrics, so there are occasional references to shorter study periods in the paper, including the study made a year later to see how I/O traffic patterns had changed.

The remainder of this paper is organized as follows. Section 2 reviews the Sprite caching system and the algorithm used to maintain consistency of client caches. Section 3 presents results on the cache consistency overhead. Section 4 presents measurements of the effectiveness of the caching system during normal system activity. Section 5 shows how variable-sized caches dynamically adapt to clients and servers of different memory sizes. Section 6 describes related work, and Section 7 concludes the paper.

Summary of Caching Measurements	
Files requiring consistency callbacks	< 1% opens
Files concurrently read shared	36% opens
Average client cache sizes	17%-35% memory
Average server cache sizes	25%-61% memory
Average client read miss ratios	35% bytes
Average client write traffic ratios	52% bytes

Table 1. Summary of results. The first part of the table contains cache consistency related figures based on open operations. The second part contains gives average cache sizes. The third part gives cache effectiveness figures based on I/O rates.



## 2. The Sprite Caching System

This section reviews the Sprite caching system originally described in [Nelson88b]. The important properties of Sprite's caching system are: 1) diskless clients of the file system use their main memories to cache data, 2) clients use a delayed-writing policy so that temporary data does not have to be written to the server, and 3) the servers guarantee that clients always get data that is consistent with activity by other clients, regardless of how files are being shared throughout the network. Servers also cache data in their main memory and use delayed writes, and the implementation of the client and server caches is basically the same.

The key characteristics of the Sprite consistency scheme are that 1) the server sees all open and close operations by clients, 2) a version number is associated with each file and incremented when a file is opened for writing, and 3) a file is not cachable on clients when it is concurrently open for writing on one client and for reading and/or writing on another client. The last point is a key simplification. If a file is concurrently write-shared by different clients, I/O operations on that file bypass the client caches and are serialized in the server cache. This scheme is based on the assumption that concurrent write-sharing is rare, although there was one application in our network that made heavy use of a shared database file. The effects of this database will be described in Sections 3 and 4.

In order to provide a consistent view of file data, the file servers track open and close operations and keep state about how their files are being cached by clients. Servers issue cache control messages to clients at open time, if needed, so that clients always get the most up-to-date file system data. A file server issues a write-back command to clientA if clientB opens a file and clientA has the most recent version of the file still dirty in its cache. Because clients use a delayed write policy, this can occur if a file is generated on one client and used by another within the 30-second aging period. A file server issues a disable caching command to clientA if clientB opens a file for writing and clientA still has the file open. The result of an open operation indicates to the opening client whether or not it can cache the file. In addition, servers increment a per-file version number each time the file is opened for writing, and clients use the version number to detect stale data in their cache.

The use of delayed writes reflects a tradeoff between reliability and performance. Sprite has a recovery system that recovers from server and client failures [Welch89], although a power failure on a client can result in the loss of recently generated data. This is no worse than a timeshared UNIX system. Our editors and source code control programs use a system call to force files through to the server's disk. Even still, there is a large amount of temporary data that is deleted before being written back to the server, as shown in Section 4.

## 3. Sharing and Consistency Overhead Measurements

The amount of file sharing and the consistency-related traffic was measured on the file servers by instrumenting the procedure that checks cache consistency and issues callbacks to clients. The results from a 36-day study are given in Table 2. The table is organized to show both hourly rates and per-server rates. The large number of opens done at night result from the nightly dumps. The various cases in the table are explained below. Note that the measurements in this section are in terms of files opened, not bytes transferred. Measurements presented in the next section indicate how much I/O traffic there is to uncachable files and what the cache hit ratios are.

File Sharing and Cache Consistency Actions							
Hour or Server	Num Opens	Non-File	Can't Cache	Read Sharing	Last Writer	Server Action	
						Write-back	Invalidate
8	6,932,578	35%	7.18%	39%	12%	0.34%	0.14%
11	1,252,039	20%	8.71%	34%	9%	0.19%	0.13%
14	1,800,274	17%	7.86%	40%	10%	0.35%	0.22%
17	2,857,620	19%	7.45%	34%	10%	0.42%	0.31%
20	1,940,819	17%	7.85%	30%	13%	0.26%	0.20%
Mint	6,976,180	10%	14.69%	44%	12%	0.17%	0.07%
Allspice	4,784,280	46%	0.04%	27%	12%	0.58%	0.01%
Oregano	1,746,430	38%	0.98%	67%	9%	0.03%	0.89%
Assault	486,320	34%	0.33%	17%	3%	1.52%	0.08%
Combined	13,993,210	27%	7.47%	37%	11%	0.34%	0.15%

Table 2. Cache consistency statistics including the number of files opened, read sharing, reuse of dirty files, and cache consistency actions. The top-half of the table gives an hourly breakdown of all the servers combined. The bottom half gives the total breakdown for each server individually. The last row has the totals for all the servers combined, which represents the total file system traffic. The data was taken over a 36-day period. Mint is a Sun3-180 with 16 Meg of memory. It serves the root directory, system log files, some shared database-like files, and most libraries and commands. Oregano is a Sun3-140 with 16 Meg of memory. It serves /tmp, some commands, and some source directories. Allspice is a Sun4-280 with 128 Meg of memory. It serves most user files, most source directories, and the swap directories used for VM paging. Assault is a DECstation 3100 with 24 Meg. It serves a few user directories.

#### Non-File

This value indicates the number of directories, symbolic links, and swap files that were opened. These files are not cached on the clients. Swap files are not cached so that VM pages really leave the machine upon page-out. Directories and links are not cached on clients because servers do all pathname evaluation [Welch86].<sup>1</sup>

#### Can't Cache

This value indicates the percentage of files opened that could have been cached on clients but were not cachable because of concurrent write sharing. The large amount of sharing measured on Mint, 15%, is explained below. The other servers see very few concurrently write shared files.

#### Read Sharing

This value counts the number of files that were open for reading by more than one process at a time, either on the same or different clients. This case is relatively frequent because of shared executable files; it happens in about 37% of the cases. It is more frequent on Mint and Oregano, the servers for the commands directories.

<sup>1</sup> There are potential benefits of caching name translations on clients, but this would introduce two sources of complexity that we wished to avoid. First, data cache consistency depends on the servers seeing each open and close so they can track file usage, and this is more complex if clients cache name translations. Second, the name translation caches would also require a consistency mechanism.

### Last Writer

This value counts the files that were written to a client's cache and then re-read or re-written by the same client before the 30-second delayed write period expired. File servers do not issue write-back commands in this case. Each of the servers, except Assault, sees a significant amount of this case, about 11% overall. This percentage is quite close to the percentage of files open for writing and suggests that most data is re-read or re-written shortly after it is generated. (85.2% opens were read-only, 9.3% opens were write-only, and 5.5% opens were for read-write access. See [Welch90] Appendix B, Table B-3.) Mint, for example, has log files that can be repeatedly updated by the same client. Oregano serves ``/tmp'', and compiler and editor temporaries account for the reuse of dirty files. Allspice has the system source directories, and compiler output usually gets re-read by the linker. Assault is too lightly loaded to experience much of this behavior.

### Server Action

This value indicates how often the servers had to issue cache control messages. ``Write-back'' indicates how many times the last writer of a file was told to write its version back to the file server. ``Invalidate'' indicates how many clients had to stop caching a file they were actively using because it became concurrently write-shared after it was opened. Write-backs happen in less than 1% of the cases, which indicates that sequential write-sharing (within the delay period) between clients is rather rare. Invalidations are also rare, except on Oregano as described below. These measurements are consistent with trace data studied by Thompson [Thompson87] who found relatively little write sharing among different users.

Two anomalies stand out in Table 2. The first is that almost 15% of the files opened on Mint were uncachable files. After some sleuthing, this value was traced to the host load database. The database was kept open on each client by a daemon that updated the client's entry once a minute. The database was kept open intentionally to make it uncached on clients and prevent the database from moving among client caches as different daemons made their updates. However, some other process apparently opens the database periodically (the daemon opens it once and keeps it open).

The second anomaly in Table 2 is the relatively large number of invalidation commands issued by Oregano. These are due to a temporary file used by pmake, our parallel compilation tool that uses process migration. pmake generates a temporary file containing the commands to be executed on the remote host. Initially, this file is cached on the host running pmake. During migration it is open by both the parent (pmake) and the child (a shell that will execute the commands on the remote host). These processes share a read-write I/O stream that the parent used to write the file and the child will use to read it. When the child migrates to the remote host the file server detects this as a case of concurrent write sharing and issues a write-back and invalidate command to the host running pmake. If the parent closed the file before the migration this would appear as sequential write sharing and contribute to the ``Write-back'' column instead.

Since the first study was made, the function of the load average database was reimplemented by an active server process, or pseudo-device [Welch88]. The server can make more intelligent choices for migration, plus its interface is more efficient [Douglass90]. Table 3 shows consistency related statistics after this change. There is a small rate of opens to uncachable files, but nothing like the 7.5% overall rate found in the earlier data.

New File Sharing and Cache Consistency Actions							
Hour or Server	Num Opens	Non-File	Can't Cache	Read Sharing	Last Writer	Server Action Write-back	Invalidate
8	934,566	47%	0.21%	36%	1%	0.75%	0.10%
11	165,476	21%	0.61%	47%	5%	0.98%	0.28%
14	280,043	22%	0.45%	46%	8%	0.81%	0.32%
17	346,710	40%	0.30%	35%	7%	0.90%	0.17%
20	211,317	26%	0.30%	48%	6%	0.96%	0.16%
Allspice	1,542,140	37%	0.32%	45%	4%	0.65%	0.19%
Anise	214,013	40%	0.05%	2%	7%	1.28%	0.02%
Assault	181,964	46%	0.49%	32%	7%	1.83%	0.15%
Combined	1,938,117	38%	0.31%	40%	4%	0.83%	0.16%

Table 3. Consistency data from Allspice, Assault, and Anise for one week in January 1991. Allspice is the primary server, while Assault and Anise store user files. The top-half of the table gives an hourly breakdown of all the servers combined. The bottom half gives the total breakdown for each server individually. The last row has the totals for all the servers combined.

#### 4. Measured Effectiveness of Sprite File Caches

This section presents results on the I/O traffic of the clients and servers, and it shows how effective the caches are during normal system use. Traffic between applications and the cache is compared with network traffic, a breakdown of the network traffic is given, and the traffic to the servers' caches is compared with the servers' traffic to their disks.

##### 4.1. Client Read Traffic

This section compares I/O traffic from applications to the cache with network traffic generated by the clients. The data presented is based on a 36-day study in November and December, 1989. The tables present I/O rates in bytes per second, the miss ratio of the cache, and a breakdown of the network traffic in terms of cache misses, uncachable data, and paging data from the VM system. In particular, there are two percentages associated with cache misses. The first percentage is the miss ratio, which is computed as follows:

$$\text{Miss Ratio} = \frac{M + U}{C + U}$$

Where  $M$  is the rate that data is fetched into the cache because of misses,  $U$  is the rate that uncachable data is read, and  $C$  is the rate that data is read from the cache by applications. The fact that  $C$  does not include  $U$  is because uncachable traffic bypasses the cache and the rates were monitored separately. In the tables below, the I/O rate in the column labeled "Cache" is for  $C$ , not  $(C+U)$ .

The second percentage associated with cache misses is its contribution to the total network traffic:

$$\text{Miss Traffic} = \frac{M}{M + U + V}$$

where  $V$  represents paging traffic from the VM system. The tables also give the proportion of uncachable data and VM data, as well as the total remote I/O rates.

Table 4 through Table 6 give the read traffic for the DECstation and Sun-3 clients. The tables are broken down into different time periods based on the time data was collected, where each row indicates the average I/O rates over the preceding interval (e.g. 8pm to 8am, 8am to 11am, and so on). The bottom row gives I/O rates and percentages averaged over 24 hours. The "Cache" column gives the read rate from the cache (not counting reads to uncachable data), and the "Misses" column gives the rate at which the cache requested data from a server. The first percentage under "Misses" is the miss ratio defined above, while the second percentage is the ratio of cache misses to all remote read traffic. The other two primary sources of remote read traffic, uncachable files and page faults, are listed in the columns

DS3100 Client Read Traffic (Bytes/Seconds and ratios)								
Hr	Cache	Misses		Uncached		PageIn		Total
8	286	21%	52 28%	10 5%	124 66%	188		
11	469	62%	289 65%	10 2%	139 31%	439		
14	490	37%	173 40%	16 3%	239 55%	430		
17	897	27%	232 58%	26 6%	138 34%	398		
20	988	47%	463 83%	12 2%	81 14%	557		
TL	570	34%	188 62%	12 4%	100 33%	302		

Table 4. Hourly read traffic for DECstation 3100 clients, which have 24 Meg main memories. The first column indicates the time of day data was taken, and the other columns have I/O rates and relative percentages. The last row averages the data over the whole trace period. I/O rates are given in bytes/second. The first percentage in the "Misses" column is the cache miss rate. The second percentage is the relative proportion of cache miss traffic to other sources of network traffic. The "Uncached" and "PageIn" columns gives rates for traffic to uncachable files and traffic to swap files, respectively. The percentages in these columns indicate their relative proportion of the total remote read traffic. The "Total" column gives the total remote read traffic.

Sun-3 (12 Meg) Client Read Traffic (Bytes/Seconds and ratios)								
Hr	Cache	Misses		Uncached		PageIn		Total
8	132	21%	23 24%	6 6%	64 68%	94		
11	566	22%	104 24%	29 6%	290 68%	425		
14	887	32%	270 46%	30 5%	280 47%	584		
17	969	25%	229 56%	20 5%	153 37%	405		
20	678	23%	147 51%	14 5%	126 43%	288		
TL	434	35%	144 43%	15 4%	167 51%	328		

Table 5. Hourly read traffic for Sun-3 clients with 12 Meg main memory.



Sun-3 (8 Meg) Client Read Traffic (Bytes/Seconds and ratios)								
Hr	Cache	Misses		Uncached		PageIn		Total
8	145	45%	57 40%	15 10%	68 47%			142
11	205	43%	77 21%	23 6%	249 69%			360
14	485	40%	179 37%	25 5%	262 55%			475
17	725	41%	283 36%	25 3%	410 52%			778
20	522	42%	215 46%	14 3%	228 49%			466
TL	322	40%	122 37%	14 4%	185 55%			330

Table 6. Hourly read traffic for Sun-3 clients with 8 Meg main memory.

labeled "Uncached" and "PageIn". The last column gives the total remote I/O traffic. Some of the total remote I/O traffic (up to 1% or 2%) is due to remote device and remote window access, which is not shown in the table.

The overall read miss rates are around 35%, with the 8 Meg Sun-3s slightly worse at a 40% miss rate. The hourly average miss rates range from about 20% to 60%, with lower percentages indicating more effective caches. Note that VM paging traffic accounts for slightly more network read traffic than cache misses. The VM traffic includes page faults on program image files, as well as faults on swap files. The DECstations, which all have 24 Meg of main memory, have the lowest paging traffic. Note that the workstations with larger memories have larger read rates to their cache, but all the workstations have about the same overall network read traffic. The larger memories reduce paging and allow for larger, more effective file caches.

Initial measurements of the read traffic highlighted a number of clients with abnormally high traffic to uncachable data. The traffic for these clients is given in Table 7. Their poor miss rate, almost 60%, stems from a single application that scanned a shared database periodically (5K to 10K every 15 seconds!). The database recorded host load averages and was used

Abnormal Client Read Traffic (Bytes/Seconds and ratios)							
Hr	Cache	Misses		Uncached		PageIn	
8	265	62%	55 13%	290 70%	66 15%		414
11	653	48%	164 20%	293 36%	336 42%		797
14	609	52%	170 23%	306 42%	244 33%		725
17	1625	34%	345 24%	336 24%	705 50%		1391
20	980	49%	332 42%	294 37%	158 20%		788
TL	516	57%	143 22%	358 56%	128 20%		633

Table 7. Hourly read traffic for a collection of DS3100 and Sun-3 clients that have abnormally high read traffic to uncachable data. The traffic stems from an application that periodically scans a heavily shared (and therefore uncached) database. Note that this collection of clients has the largest I/O rates to its cache and that the miss rate does not include traffic to uncachable data, only to data that could be cached.

to choose hosts for process migration. A daemon process on each host updated the database once a minute, and it kept the database open to force it to be uncachable. An X widget that displayed the number of hosts available for migration caused periodic scans of the database, and the effect on network traffic was significant. The process migration system has been changed to use a server process to manage host selection instead of using a shared file. Table 8 shows the read traffic for a collection of DECstation and Sun-4 clients after this change was made. There is almost no read traffic to uncachable data, and VM paging traffic dominates the network.

#### 4.2. Client Write Traffic

Table 9 through Table 11 give the write traffic for the DECstation and Sun3 clients. The format of the tables is similar to those with read traffic. Hourly breakdowns are given, and remote traffic is divided among cache misses, uncachable data, and writes due to page outs.

New Client Read Traffic (Bytes/Seconds and ratios)							
Hr	Cache	Misses		Uncached		PageIn	Total
8	68	23%	13 14%	2 3.0%	80 82%		96
11	375	53%	196 32%	7 1.3%	399 66%		604
14	590	37%	211 28%	11 1.6%	514 69%		738
17	736	37%	266 32%	10 1.3%	547 66%		824
20	275	30%	80 18%	6 1.5%	341 79%		428
23	276	35%	91 29%	11 3.8%	210 67%		313
TL	174	35%	59 28%	3 1.9%	149 70%		213

Table 8. Read traffic for a collection of DECstation and Sun-4 clients after the shared database was replaced by a server process. There is almost no read traffic to uncachable data, and VM paging traffic dominates the network traffic.

DS3100 Client Write Traffic (Bytes/Seconds and ratios)						
Hr	Cache	WriteBack		Uncached	PageOut	Total
8	146	48%	63 49%	14 11%	48 38%	127
11	174	41%	69 46%	5 3%	74 49%	150
14	270	55%	146 65%	5 2%	69 31%	222
17	530	44%	230 73%	6 2%	74 23%	314
20	333	47%	154 80%	5 3%	29 15%	190
TL	244	51%	120 63%	10 5%	57 30%	189

Table 9. Hourly write traffic for all the DECstation 3100 clients. The format of the table is the same as Table 4. An uncached database is updated once a minute by each host, and this creates a small amount of uncachable I/O traffic. Note that the overall write miss rate is 51%, meaning that half the data written to the cache is not written out.

Sun-3 (12 Meg) Client Write Traffic (Bytes/Seconds and ratios)									
Hr	Cache	WriteBack			Uncached		PageOut		Total
8	69	52%	30	53%	11	20%	13	23%	57
11	148	50%	71	49%	7	5%	61	43%	143
14	267	57%	150	54%	8	3%	79	28%	273
17	327	53%	170	67%	7	3%	70	28%	251
20	239	37%	84	55%	7	4%	56	37%	151
TL	148	52%	74	53%	8	6%	45	32%	140

Table 10. Hourly write traffic for all the 12 Meg Sun-3 clients. The write traffic is similar to that of the DECstation clients.

Sun-3 (8 Meg) Client Write Traffic (Bytes/Seconds and ratios)									
Hr	Cache	WriteBack			Uncached		PageOut		Total
8	86	53%	38	46%	15	19%	27	33%	83
11	134	61%	77	46%	11	6%	77	46%	167
14	186	53%	94	49%	11	6%	82	43%	190
17	354	69%	244	67%	9	2%	104	29%	360
20	172	59%	98	41%	9	3%	125	53%	236
TL	154	59%	85	49%	12	7%	73	42%	173

Table 11. Hourly write traffic for all the 8 Meg Sun-3 clients. The write miss rate is slightly worse (higher) in comparison with the DECstation and 12 Meg Sun-3 clients.

The effectiveness of the cache in trapping short lived data is given by the traffic ratio:

$$\text{Traffic Ratio} = \frac{W + U}{C + U}$$

Where  $W$  is the amount of data written out of the cache to a server,  $C$  is the amount of data written into the cache by applications, and  $U$  is the amount of data written to uncachable files.

The most notable result in the tables is that the writeback traffic ratio ranges from about 40% to 60%, averaging 52% overall. This means that about half the data written by applications was removed or overwritten in the 30 second aging period. Traffic to uncachable data accounts for only a few percent of the network traffic, although it does increase the miss ratio by a couple percent. Page out traffic accounts for less than half the network traffic, and it is as low as 30% of the traffic from the DECstation 3100s. Table 12 shows more recent data for the Sprite network, after the uncachable migration database was replaced with a server. There is still evidence of some shared files that are updated steadily, but the I/O rate is much smaller. Shared files are still used to record user logins and to log system events. Note that the write traffic ratio is 70%, well above the 50% found in the initial study. This follow-on study, however, was only one week long and occurred during a period of relative inactivity as shown by the overall I/O rates. It was probably not long enough to match the accuracy of the earlier study with regards to long term cache effectiveness.

New Client Write Traffic (Bytes/Seconds and ratios)						
Hr	Cache	WriteBack		Uncached	PageOut	Total
8	28	72%	20 71%	1 5.0%	5 19%	28
11	111	84%	94 77%	1 1.4%	21 17%	121
14	176	72%	127 60%	1 0.8%	72 34%	210
17	259	66%	173 57%	1 0.5%	120 39%	301
20	85	66%	56 32%	1 0.9%	109 63%	171
23	68	69%	47 48%	1 1.7%	32 33%	97
TL	56	70%	39 56%	1 1.3%	26 38%	69

Table 12. Hourly write traffic after the shared migration database was replaced with a server process. The clients are a collection of Sun-4s and DECstations. The rate of I/O to uncachable files has dropped significantly, but there is still evidence of uncachable files, most likely system log files and the user login database.

### 4.3. Server I/O Traffic

This section presents the I/O traffic from the standpoint of the file servers. In the case of a file server it is interesting to compare the traffic to its cache to the traffic to its disks. Two metrics are given, the "File Traffic" and the "Metadata Traffic." *Metadata* is data on the disk that describes a file and where it lives on disk. This includes the descriptor that stores the file's attributes, and the index blocks used for the file map. The "File Traffic" represents I/O to file data blocks as opposed to the metadata information. The combination of file traffic and metadata traffic gives the total disk traffic for the file server. Three ratios are given that compare cache traffic to disk traffic:

$$\text{File Traffic Ratio} = \frac{F}{C}$$

$$\text{Metadata Traffic Ratio} = \frac{M}{C}$$

$$\text{Total Traffic Ratio} = \frac{M + F}{C}$$

Where  $F$  is the disk traffic from the cache to file data,  $M$  is the disk traffic to metadata, and  $C$  is the traffic between applications and the cache. There are no uncachable files on a file server. The file traffic ratio ignores the effects of metadata, while the total traffic ratio includes it.

The server I/O traffic from a 20-day study period, October 29 through November 19, 1989, is given in Table 13 and Table 14. Figure 1 shows the server I/O traffic for a combination of all servers averaged over a 6-month study period. Servers were sampled every hour, 24 hours a day. The graphs indicate that the server caches are effective for reads during peak usage hours. The server caches are less effective for writes because client caches trap out most of the short-lived data. The write graph highlights the large amount of write traffic to metadata. File descriptors have to be updated with access and modify times, so just reading a file ultimately causes its descriptor to be written to disk. Furthermore, the 128-byte descriptors are written 32 at time in 4K blocks so there is extra traffic from unmodified descriptors. The large

Server I/O Traffic (Bytes/Second)								
Host		Cache Traffic		File Traffic		MetaData Traffic		Total Disk
		bytes/s	(dev)	bytes/s	(dev)	bytes/s	(dev)	bytes/s
Mint	r	8601	(9321)	3427	(7876)	283	(982)	3710
	w	921	(640)	863	(507)	4133	(1913)	4996
Oregano	r	4421	(9129)	3201	(8242)	453	(1101)	3654
	w	932	(3163)	804	(2994)	1295	(1295)	2099
Allspice	r	11478	(18313)	5970	(16398)	520	(2062)	6490
	w	5174	(9495)	3838	(6142)	1692	(2013)	5530
Assault	r	1808	(7342)	1481	(7128)	180	(604)	1661
	w	529	(3081)	291	(1385)	350	(676)	641
combined	r	25946	-	13932	-	1433	-	15515
	w	7305	-	5620	-	7481	-	13266

Table 13. I/O traffic on the file servers over a 20-day period. The upper row for each server gives read I/O rates, the lower row gives write rates. The average and standard deviation are given for the bytes/sec transferred to and from the cache ("Cache"), file data blocks on disk ("File"), from file descriptors and index blocks on disk ("MetaData"), and the total traffic to the disk ("Total").

Server I/O Traffic (Megabytes and Ratios)							
Host		Cache	File	MetaData	Total Disk		
Mint	r	15172	6046 40%	500 3%	6546	43%	
	w	1624	1523 94%	7291 449%	8814	542%	
Allspice	r	18861	9811 52%	854 5%	10665	57%	
	w	8503	6307 74%	2781 33%	9088	107%	
Oregano	r	8199	5937 72%	839 10%	6776	83%	
	w	1729	1491 86%	2402 139%	3893	225%	
Assault	r	3131	2565 82%	311 10%	2876	92%	
	w	917	505 55%	606 66%	1111	121%	
combined	r	45364	24358 54%	2505 6%	26863	59%	
	w	12772	9826 77%	13080 102%	22906	179%	

Table 14. This gives the total megabytes transferred for the results given in Table 13, and the percentage that this is of the megabytes transferred to or from the cache. The total disk traffic can be greater than 100% of the cache traffic because of metadata traffic.

amount of metadata traffic at 2am is because the UNIX tar program is used for our nightly dumps, and it changes the access time of every directory and any files that were dumped. The effect of metadata changes has been noted by Hagmann[Hagmann87], who converted the Cedar file system to log metadata changes, which reduced metadata traffic considerably. Current research in the Sprite group involves a log-structured file system where all data, file and meta-data, is logged [Ousterhout89].

The tables show that the cache on Mint, the root server, is effective in eliminating reads (40% file traffic in the 20-day study), but not as good at eliminating writes (94% traffic ratio in the 20-day study). Its read hits occur on frequently-used program images and the load average database. The steady updates to this file and other logs trigger a steady update of disk



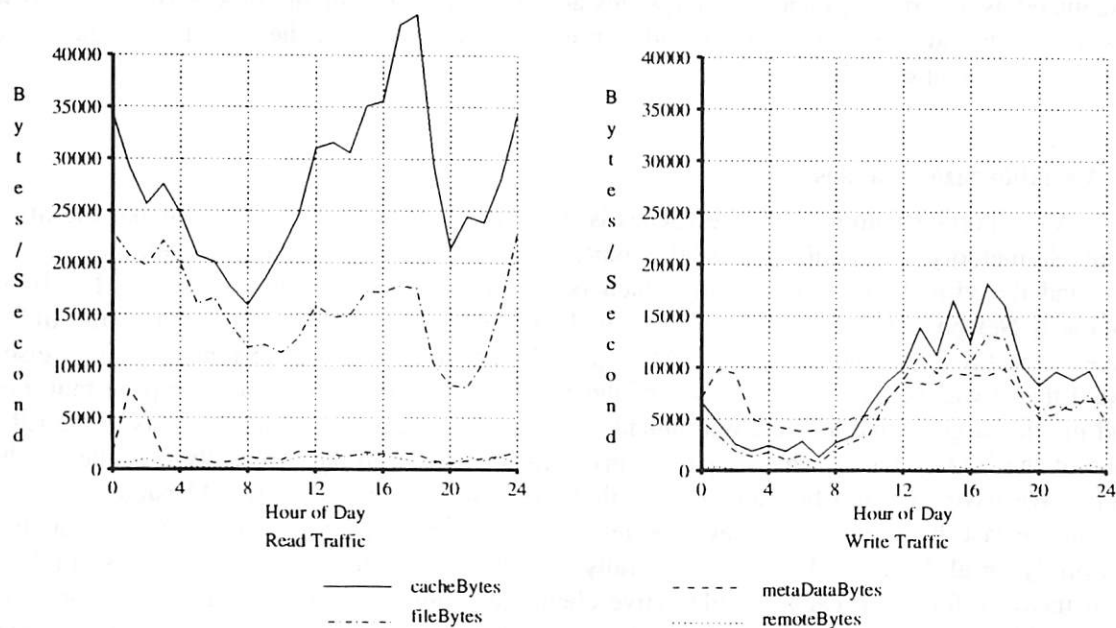


Figure 1. Server I/O traffic averaged from July 8 to December 22, 1989. The left-hand graph has read traffic and the write-hand graph has write traffic. "cacheBytes" are file data bytes transferred between the cache and remote clients or server-resident applications. "fileBytes" are file data bytes transferred between the disk and the cache. "remoteBytes" are file data bytes accessed remotely by server-resident applications. "metaDataBytes" are metadata bytes transferred to and from the disk. The total disk traffic is the sum of "fileBytes" and "metaDataBytes".

descriptors, which explains Mint's high metadata traffic.

The 20-day study reported in the tables was made after a bug was fixed that prevented continuously updated files from being written through to disk. With this bug present, Mint's traffic ratio was about 50% for file data, while the other servers had write traffic ratios of 70% to 80% or more. Mint's low traffic ratio prompted a search for a bug in the cache write-back code, and this 20-day study was made after it was fixed. That Mint's current traffic ratio changed from 50% to 94% indicates almost half the data written to it is to continuously updated files, i.e. the migration database. The 6 month study graphed in Figure 1, however, includes both versions of the system so there actually should have been slightly more file write traffic to update the migration database.

Allspice and Oregano are directly comparable because they store the same type of files (many files were shifted from Oregano to Allspice during the 20-day study period). Allspice's cache is about 10 times the size of Oregano's and it is clearly more effective. This is to be expected because the server's cache is a second-level cache, with the clients' caches being the first level. The server's caches have to be much larger than the client's caches because the locality of references to their cache is not as good.

It is also interesting to see how the caches skew the disk traffic towards writes. During the 20-day study period, the traffic to the server caches was about 22% writes. The traffic to

the server disks was 26% metadata writes and 20% data writes. If the metadata traffic is discounted as an artifact, then the data writes accounted for 40% of the disk traffic. The skew towards writes at the disk level should continue as the server caches get larger and more effective at trapping reads.

## 5. Variable-Sized Caches

An important feature of Sprite caches is that they vary in size in order to make use of all available memory. Nelson[Nelson88a] explored ways of trading memory between the file system and the virtual memory system, which needs memory to run user programs. The basic approach Nelson developed was to compare LRU times (estimated ages) between the oldest page in the FS cache and the oldest VM page and pick the oldest one for replacement. Nelson found that it was better to bias in favor of the VM system in order to reduce the page fault rate and provide a good interactive environment. The bias is achieved by adding a bias to the LRU time of the VM system so that its pages appear to be referenced more recently than they really were. We have chosen a bias against the file system of 20 minutes. Any VM page referenced within the last 20 minutes will never be replaced by a FS cache page. This policy is applied uniformly on all hosts, and it adapts naturally to both clients and servers. Servers use most of their memory for a file cache, while active clients use most of their memory to run user programs. Idle clients become hosts for process migration, and large idle programs (i.e. the window system) tend to get paged out and replaced by more file cache as well as the migrating applications.

### 5.1. Average Cache Sizes

Table 15 gives the average and maximum cache sizes as measured over the full 6-month study period. The file servers are listed individually. The clients are grouped according to the amount of physical memory and processor type of the host, and the results are averaged. The adaptive nature of the cache sizes is evident when comparing clients and servers with the same memory size; the file servers devote more of their memory to the file cache. This difference is not achieved via any special cases in the implementation, but merely by the uniform application of the 20-minute bias against the file system described above.

The cache occupies a larger percentage of main memory as the memory size increases, indicating that the extra memory is being utilized more by the file cache than by the VM system. This trend is most evident on the Sun3 clients, of which there is a good population size, and in which the Sprite implementation is the most mature. Doubling the physical memory on a Sun3 client quadruples the average cache size on the client; it increases from 17% to 34% of the physical memory.

The variability of the client caches is indicated by the standard deviation and the maximum observed values. The variability tends to increase as the memory gets larger, indicating that the cache is trading more memory with the VM system. The DECstations have lower variability because their cache was limited to about 8.7 meg during most of the study period. (This limitation was a software limit that was eventually removed.)

The results from the servers show that Mint and Oregano can only devote on average a little over half their memory to their file cache, yet their maximum cache size is as much as 3/4 of their memory. Their caches ramp up to a maximum shortly after booting, and then

Cache Size (Megabytes)					
Host	Mem	Average		Std Dev	
Allspice*	128	67.8	52%	22.14	17%
Assault**	24	7.5	31%	4.55	19%
Mint	16	9.0	56%	1.23	8%
Oregano	16	8.6	54%	1.77	11%
Sun3	8	1.4	17%	0.96	12%
Sun3	12	3.2	27%	1.76	15%
Sun3	16	5.5	34%	2.86	18%
Sun4	12	2.1	17%	1.73	14%
Sun4	24	6.0	25%	3.70	15%
DS3100	24	6.3	26%	2.27	9%

Table 15. Cache sizes as a function of main memory size and processor type, averaged over the study period. The average, standard deviation, and maximum values of the observed cache sizes are given. The sizes are megabytes and percentage of main memory size. The file servers are listed individually. The rest of the clients are averaged together based on CPU type and memory size.

\* Allspice's cache was limited to at most 78.13 Mbytes.

\*\* Assault's cache was limited to 8.7 Meg during most of the study.

gradually decline in size as the kernel builds up state information about how its files are being used. The servers also grow the number of RPC server threads they keep, and each thread has a significant amount of pre-allocated buffer space as well as a kernel stack.

In the case of Allspice, however, the limitation on its cache size is due to an artifact of the memory mapping hardware on the Sun4. The file cache uses hardware page map entries, and if the cache gets too large it can cause extreme contention for the few remaining map entries. Its cache is fixed arbitrarily at about 80 meg because of this. Even with this limitation, the cache on Allspice is 10 times the size of the cache on Mint and Oregano, and measurements presented in Section 4.3 indicate that a server needs a large cache like this for the cache to be really effective.

## 6. Related Work

The Sprite caching system was initially studied by Nelson for his thesis work [Nelson88a]. He compared 9 different client writing policies in combination with 4 different server writing policies on a set of benchmarks. He found that using a delay policy on both clients and servers minimized I/O traffic and provided the best client response time. In contrast, a "write through on close" policy, which is used in NFS, increases network and disk traffic and causes clients to wait at close time.

AFS uses a hybrid strategy of caching temporary files (those under "/tmp") with a delay policy, but writing through other files at close. Nelson found this to be almost as good as the Sprite policy in terms of network bandwidth reduction, although significant delays can still occur when non-temporary data is written through to the server. AFS caches remote files on the local disk, in contrast to Sprite's use of main memory. Howard found a read miss rate of

about 20% on the AFS client caches [Howard88], which is better than the 35% miss rate found in this study. The better miss rate in AFS is because the disk-based caches of AFS are larger than the main memory caches in Sprite (AFS clients usually have 20 or 40 Meg caches). The response time and server CPU utilization of NFS, AFS, and Sprite were compared in [Nelson88b]. Sprite provided the best response time (25% faster than AFS and 35% faster than NFS), while AFS had better (i.e. lower) server utilization (16% server CPU utilization under load for AFS vs. 38% for Sprite and 80% for NFS). Another comparison between Sprite and NFS can be found in [Srinivasan89], in which the addition of the Sprite delayed-write policy and consistency mechanism to an NFS system improved performance significantly.

Earlier studies of I/O traffic include Ousterhout's measurements of timesharing VAXes running 4.2 BSD UNIX [Ousterhout85], which reported per-user I/O rates of 300-600 bytes/sec when averaged over 10 minute intervals, and rates of 1400 to 1800 bytes/sec when averaged over 10 second intervals. These rates do not include paging traffic, and they are for active users only. Rates are higher over shorter intervals because there are fewer active users in a shorter interval. The rates obtained for Sprite clients, about 1500 bytes/sec for combined read and write traffic in the mid-afternoon, are averaged over 180 minutes and include periods of inactivity.

## 7. Conclusion

This paper has reviewed the Sprite caching system and reported on its performance when supporting day-to-day work in our user community. There are a number of significant results. Client write traffic ratios average about 50%, meaning that half the data generated by applications is never written through to the server because it is deleted or overwritten before the 30-second aging period expires. Client read miss rates are about 35%, indicating reasonable effectiveness. There is low overhead from consistency-related actions. In less than 1% of open operations did the server have to issue cache control messages. The most interesting negative result is that the shared database used to record host load averages accounted for approximately 10% of the network write traffic and up to 60% of the network read traffic for some clients. This problem has been cured by replacing this heavily shared file with a network server process and tuning the interface to it. The result is that concurrent write sharing, which causes files to be uncacheable on clients, occurs in less than 1% of the files opened. There is very little consistency-related traffic between the servers and clients, and (after fixing one application!) there is very little data traffic to uncacheable data.

## 8. References

- Douglis90. F. Douglis, "Transparent Process Migration for Personal Workstations", PhD Thesis, Sep. 1990. University of California, Berkeley.
- Hagmann87. R. Hagmann, "Reimplementing the Cedar File System Using Logging and Group Commit", *Proc. of the 11th Symp. on Operating System Prin.*, Nov. 1987, 155-162.
- Howard88. J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham and M. J. West, "Scale and Performance in a Distributed File System", *Trans. Computer Systems* 6, 1 (Feb. 1988), 51-81.

- Nelson88a. M. N. Nelson, "Physical Memory Management in a Network Operating System", PhD Thesis, Nov. 1988. University of California, Berkeley.
- Nelson88b. M. Nelson, B. Welch and J. Ousterhout, "Caching in the Sprite Network File System", *Trans. Computer Systems* 6, 1 (Feb. 1988), 134-154.
- Ousterhout85. J. Ousterhout, H. D. Costa, D. Harrison, J. Kunze, M. Kupfer and J. Thompson, "A Trace-Driven Analysis of the UNIX 4.2 BSD File System", *Proc. 10th Symp. on Operating System Prin., Operating Systems Review* 19, 5 (December 1985), 15-24.
- Ousterhout88. J. Ousterhout, A. Cherenon, F. Douglass, M. Nelson and B. Welch, "The Sprite Network Operating System", *IEEE Computer* 21, 2 (Feb. 1988), 23-36.
- Ousterhout89. J. Ousterhout and F. Douglass, "Beating the I/O bottleneck: A Case for Log-Structured File Systems", *Operating Systems Review* 23, 1 (Jan. 1989), 11-28.
- Srinivasan89. V. Srinivasan and J. Mogul, "Spritely NFS: Experiments with Cache-Consistency Protocols", *Proc. 12th Symp. on Operating System Prin., Operating Systems Review* 23, 5 (December 1989), 45-57.
- Thompson87. J. Thompson, "Efficient Analysis of Caching Systems", PhD Thesis, 1987. University of California, Berkeley.
- Welch86. B. B. Welch and J. K. Ousterhout, "Prefix Tables: A Simple Mechanism for Locating Files in a Distributed Filesystem", *Proc. of the 6th ICDCS*, May 1986, 184-189.
- Welch88. B. B. Welch and J. K. Ousterhout, "Pseudo-Devices: User-Level Extensions to the Sprite File System", *Proc. of the 1988 Summer USENIX Conf.*, June 1988, 184-189.
- Welch89. B. Welch, M. Baker, F. Douglass, J. Hartmann, M. Rosenblum and J. Ousterhout, "Sprite Position Statement: Use Distributed State for Failure Recovery", *Proc. of the Second Workshop on Workstation Operating Systems (WWOS-II)*, Sep. 1989, 130-133.
- Welch90. B. B. Welch, "Naming, State Management, and User-Level Extensions in the Sprite Distributed File System", PhD Thesis, 1990. University of California, Berkeley.



1. The first part of the paper discusses the background and motivation for the research.	1. The first part of the paper discusses the background and motivation for the research.
2. The second part describes the system architecture and the components involved.	2. The second part describes the system architecture and the components involved.
3. The third part presents the experimental setup and the results of the experiments.	3. The third part presents the experimental setup and the results of the experiments.
4. The fourth part discusses the conclusions and the future work.	4. The fourth part discusses the conclusions and the future work.
5. The fifth part is the references.	5. The fifth part is the references.
6. The sixth part is the appendix.	6. The sixth part is the appendix.
7. The seventh part is the bibliography.	7. The seventh part is the bibliography.
8. The eighth part is the index.	8. The eighth part is the index.
9. The ninth part is the glossary.	9. The ninth part is the glossary.
10. The tenth part is the list of figures.	10. The tenth part is the list of figures.
11. The eleventh part is the list of tables.	11. The eleventh part is the list of tables.
12. The twelfth part is the list of equations.	12. The twelfth part is the list of equations.
13. The thirteenth part is the list of symbols.	13. The thirteenth part is the list of symbols.
14. The fourteenth part is the list of abbreviations.	14. The fourteenth part is the list of abbreviations.
15. The fifteenth part is the list of acronyms.	15. The fifteenth part is the list of acronyms.
16. The sixteenth part is the list of keywords.	16. The sixteenth part is the list of keywords.
17. The seventeenth part is the list of authors.	17. The seventeenth part is the list of authors.
18. The eighteenth part is the list of reviewers.	18. The eighteenth part is the list of reviewers.
19. The nineteenth part is the list of sponsors.	19. The nineteenth part is the list of sponsors.
20. The twentieth part is the list of acknowledgments.	20. The twentieth part is the list of acknowledgments.

# Using Kernel-Level Support for Distributed Shared Data<sup>\*</sup>

David L. Cohn, Paul M. Greenawalt, Michael R. Casey, Matthew P. Stevenson  
Department of Computer Science and Engineering  
University of Notre Dame  
Notre Dame, Indiana 46556  
Contact: dlc@cse.nd.edu

## ABSTRACT

This paper investigates the use of distributed shared data as a programming paradigm for distributed applications. It describes experiences with the kernel-level support for distributed shared data available in the ARCADE distributed environment. It reports on the substantial program simplification that results from replacing a classic message passing scheme with the distributed shared data model. Also, although a carefully designed messaging implementation can result in less communication overhead, the paper shows that the shared data concept actually promotes faster execution. The key is that messaging frequently results in synchronous solutions while shared data facilitates asynchronous ones. Three example applications are discussed. The first is a simple data sharing exercise, the second does cooperative computation and the third shows how global pointer-like variables can be used to build distributed dynamic data structures.

## 1. Introduction

The distributed shared memory paradigm [1] was originally proposed to simplify the task of writing distributed algorithms. It was felt that applications would be simplified if they did not have to explicitly communicate. Subsequent work [2][3][4] has focused on improving the efficiency of distributed shared memory and dealing with the problems caused by page granularity. An alternative form of shared data is to use shared data-objects [5]. In this paper, we discuss three examples of distributed shared memory which support the claim that it simplifies coding. The examples also point out another, perhaps even more valuable, effect of using the distributed share memory model: it promotes the use of asynchronous operation. Tasks which explicitly exchange messages frequently are written to wait for each other. With the implicit communication of shared memory, it is natural to let each task run at its own pace. Thus, even though distributed shared memory can entail more overhead, it may lead to faster code.

Distributed shared memory has been proposed as a convenient method of data sharing between elements of a distributed application. The ARCADE distributed kernel has a unique view of data sharing and includes kernel-level support for it. The title of this paper mentions distributed shared

---

<sup>\*</sup> This work was supported by IBM Research Agreements 6093 and 182

*data*, rather than distributed shared *memory*. Normally, distributed shared memory refers to the paging model originally proposed by Li. Shared memory is handled much like virtual memory with page boundaries determining how updates are made. With ARCADE, the size and content of shared entities are set by the application and can be tailored to how the data is to be used.

ARCADE was originally designed as an architecture to provide interconnections of heterogeneous machines with transparent machine boundaries. Support for differing data representations is built into the kernel. This means that the kernel has to know how memory is being used in order to provide necessary data translations. With this structural information, ARCADE can offer a new model of distributed shared memory. Data is stored in structured carriers called *data units* which can be shared between machines. The kernel defines locks for data units and these allow applications to control how and when the shared data is updated.

This paper summarizes the ARCADE architecture and describes three examples of its support for distributed shared data. The first is a simple exercise which illustrates a variety of cooperation scenarios possible with shared data. It shows the value of asynchronous operation which hides communication overhead. The second is a classic calculation problem which demonstrates an asynchronous solution of LaPlace's equation. The final example uses ARCADE's support for distributed dynamic data structures to provide a model that simply doesn't exist with messaging.

## 2. ARCADE Distributed Kernel

This section introduces the ARCADE distributed kernel [6][7][8] and explains how it handles data. ARCADE is a minimal software platform for distributed applications which are viewed as collections of independent, cooperating tasks. These tasks can interact in a uniform and effective manner even when they reside on heterogeneous machines which use different data representations. This distinguishes ARCADE from other distributed kernels, such as Mach [9][10], V kernel [11], and CHORUS [12] which are not concerned with heterogeneity. ARCADE's support of heterogeneity has led naturally to sophisticated and powerful kernel-level support for distributed shared data.

ARCADE defines two basic abstractions: active *tasks* and passive *data units*. Tasks are named entities which possess an address space and an *input queue*. The names are globally unique and location independent and are used for inter-task identification. Address spaces are private; they are not shared between tasks. Input queues provide a port-like mechanism for tasks to transfer and share data.

The data unit abstraction was introduced in ARCADE to deal with problems of differing data representations. They are essentially structured carriers of data which can be mapped into the address spaces of tasks. Tasks can create and discard data units instead of allocating and deallocating memory. In addition, the data units may be transferred and shared between tasks. ARCADE defines locking primitives which enable tasks to ensure consistency of shared data units. Finally, ARCADE supports the construction of dynamic data structures within and between data units. Subsequent sections describe these data unit services in greater depth.

## 2.1 Allocation and Release

ARCADE tasks allocate data units just as normal processes allocate memory. However, in addition to specifying the *size* of the memory region desired, a task specifies the *structure* of the data that will be stored there. The structure specification is very much like that found in high-level languages such as Pascal and C. A data unit may consist of simple atomic entities (16-bit unsigned, 32-bit floating point, character, etc.) and structures built from these entities. The ARCADE kernel processes an **ALLOCATE** request by mapping an appropriately sized chunk of memory into the task's address space and returning the address of the new data unit. The data unit's structural specification is stored in a kernel structure called a *data unit descriptor*, or DUD. DUDs are not visible to tasks, but a DUD accompanies each data unit throughout its lifetime.

A data unit is unmapped from a task's address space with the **RELEASE** service. If no other tasks have access to the data unit and if it is not the target of a data unit link, the memory occupied by it and by its associated DUD will be deallocated.

## 2.2 Transfer and Sharing

ARCADE provides two fundamental task interaction mechanisms: messaging through data unit transfer and shared data by data unit sharing. Since machine boundaries are transparent, the ARCADE kernel provides full support for distributed shared data. The data unit's known structure and its lockability allow the kernel to handle many of the problems of such support.

A task uses the **SHARE** service to share a data unit with another task. Parameters to **SHARE** include the name of the target task and the address of the data unit to be shared. The kernel places the specified data unit on the input queue of the target task. Subsequently, the target task will use another ARCADE service, **ACCESS**, to gain addressability to the data unit. As a result, *both* tasks will have the same data unit mapped into their address spaces.

If both tasks reside on one machine, then **SHARE** may be implemented using physical shared memory. If, however, the target task is remote from the origin, the ARCADE kernel automatically generates a replica of the data unit on the destination machine. Furthermore, if the destination machine has a different hardware architecture than the origin machine, ARCADE uses the information in the DUD to transparently translate the data to the appropriate representation for the target machine.

## 2.3 Locking

Since the **SHARE** service allows multiple tasks to have simultaneous direct access to a data unit, ARCADE provides locks which allow tasks to synchronize access to the data. ARCADE relies on an application's use of locks to maintain consistency among the replicas of a data unit. When a task asserts a *read lock*, the kernel guarantees that no other task has a *write lock* and that the local copy of the data unit will not change. A write lock assures a task that no other task has a concurrent lock and the release of a write lock causes any changes in the data to be propagated to all replicas.

asks are not obligated to use data unit locks. Since data units are directly addressable, tasks may read and write them without first obtaining the corresponding lock. Locking may be viewed as a user level protocol which avoids the *stale data problem* described in [13]. However, other user level protocols may be employed to maintain consistent shared data. In fact, as Cheriton [14] has pointed out, there are times it is not necessary to guarantee data consistency. Judicious use of the locks allows a task to propagate and receive updates only when necessary. This permits tasks to run asynchronously and can result in substantial performance enhancement.

## 2.4 Dynamic Data Structures

Since ARCADE tasks do not share address spaces, the sharing of a data unit which contains pointer variables would pose a problem. The classic implementation of pointers as addresses is not meaningful when the pointer is shared between tasks. However, without some type of pointer, dynamic data structures can not be constructed. Therefore, ARCADE defines a new pointer-like variable type, called a *data unit link*. A data unit link identifies a data unit and points to a location within that data unit. They can be used to implement dynamic data structures which consist of multiple data units.

Data unit links are essentially system-level pointers and may not be read or written directly by tasks. Instead, tasks use the SETLINK service to assign a value to a data unit link. The ACCESS service allows a task to request that ARCADE map the target of a data unit link into the task's address space.

Although it is more expensive to manipulate data unit links than simple pointers, data unit links can be quite valuable. They allow tasks to construct and share arbitrarily complex, multi-level data structures. Such structures may even be shared between heterogeneous machines.

## 2.5 Input and Output Lines

ARCADE tasks can interact through a set of *input and output lines*. These allow a task on one machine to effect the operation of a task on another machine. A task's output lines are binary signals which can be set and reset by the task. Its input lines are also binary and can be sensed by the task. More importantly, they can determine the task's *run/sleep* and *live/die* controls. As their names suggest, these controls govern whether or not the task is dispatchable and whether or not it will continue to exist.

The task itself determines how its input lines effect its controls. It can ask the kernel to "connect" the output lines from another task to its input lines. It can also "connect" system signals such as a timer tick or a physical interrupt. The input lines pass through the software equivalent of a programmable logic array, or PAL, whose outputs are *run/sleep* and *live/die*. As with the input connections, the task determines the logic function that relates its input lines to its controls. Thus, for example, a task can sleep until an interrupt occurs or another task completes a job and the timer has ticked.



## 2.6 Prototype Implementation of ARCADE

ARCADE was originally designed as an architecture. Several prototype implementations have been built at the University of Notre Dame. The original version ran on 80286/80386-based PS/2 computers communicating over a token ring. A similar variant also was operational on the IBM System/370 architecture. The current prototype runs only on IBM 80386-based PS/2s connected via a 4 Mbps token ring. All measurements reported here are from 16 Mhz systems.

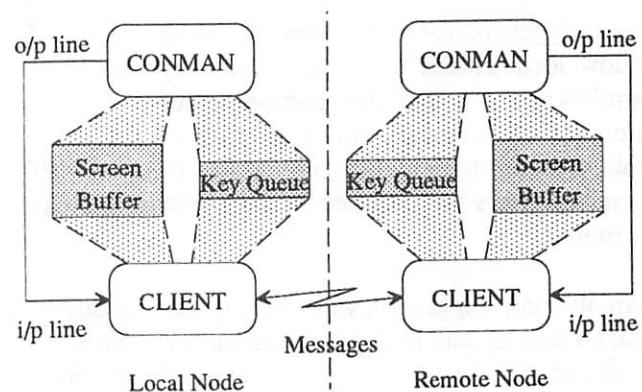
## 3. PHONE - A Party-line Application

The initial application of ARCADE's shared data support is a talk-like utility, called PHONE, that lets a group of users type messages to each other. Each user has a keyboard and screen. Whenever something is typed on one keyboard, it appears on all screens. Although some delays are unavoidable, the screens are updated as quickly as possible. Several versions of PHONE were developed to demonstrate various approaches to sharing data. This section describes the structure of these solutions and shows how they were used to measure the effectiveness of distributed shared data.

As noted above, ARCADE is a distributed kernel. Normal operating system services, such as user interfaces, file systems and so forth, are built as tasks on top of the kernel[15], in the model of Mach and Chorus. The console manager task, named CONMAN, handles keyboard and screen interfaces. A client task can ask CONMAN for a *logical console* and CONMAN will allocate a key queue data unit and a screen buffer data unit for that client. The client may subsequently ask CONMAN for access to these data units which then become shared between CONMAN and the client.

When the user *selects* the client's logical console, the client becomes the foreground task. CONMAN copies its screen buffer to the physical screen and places any subsequent keystrokes into its key queue. A foreground client can change its screen buffer directly and then ask CONMAN to update the physical screen. CONMAN places keystrokes into the key queue and then pulses one of its output lines to notify the client.

The initial version of PHONE had a local client task on each machine which interacted conventionally with its local CONMAN. Figure 1 illustrates a two node group and shows the local sharing of the key queue and screen buffer. Whenever a key was pressed, the client would read it from the key queue and send it in a message to all other client tasks in the group. Each client was responsible for properly modifying its screen buffer and asking its CONMAN to update the physical screen.



**Figure 1** Message passing implementation

To measure the performance of each approach to PHONE, a string of 100 keystrokes was generated at one node and the time to propagate

them to all other nodes was measured. The keystrokes were generated from the keyboard. Even without communication, it took 35 milliseconds per key for CONMAN to process the keys and write them to the screen. The messaging version was used as the baseline and results for up to four remote nodes are shown in Figure 2. For only one remote node, CONMAN is on the critical path and the time per key is about 35 milliseconds. With each additional node, the local client spends an additional 15 milliseconds waiting for communication. With two remote nodes, the client time dominates, and for each successive node, time increases by 15 milliseconds.

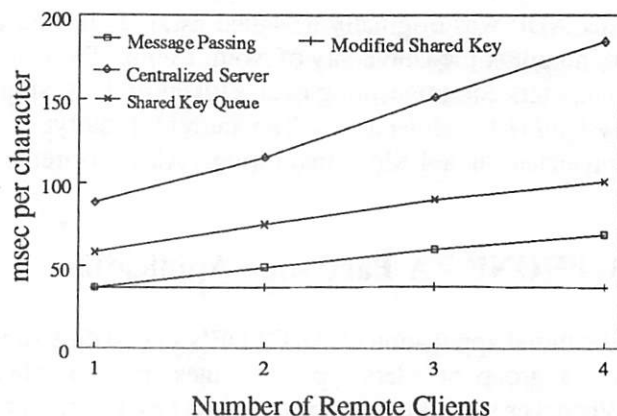


Figure 2 Performance of various schemes

The second implementation used ARCADE's support for sharing data across machine boundaries to build PHONE with only one centralized client task. As shown in Figure 3, the centralized client obtained a logical console on each node. When this console was selected, the centralized client was able to receive keystrokes from the remote machine. It determined what should be displayed on each screen, modified all of the screen buffers and notified each CONMAN to update the physical screens.

As might be expected, this approach did not yield good performance. Even with keystrokes being generated at only one node, Figure 2 shows that the added communication to lock and update remote screens significantly degraded performance. However, since the client did not have to provide message receiving routines, the code was much simpler. Not counting the initialization code for PHONE, it was 50% shorter than the code for messaging.

One suggestion for another version was to allow local clients to process local keystrokes and to update the appropriate portions of all screens. However, this entailed many writers contending for the screen buffers and resulted in a significant increase in communication to arbitrate between the writers.

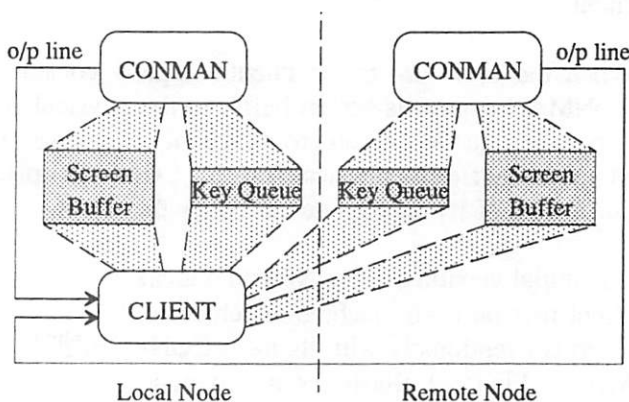
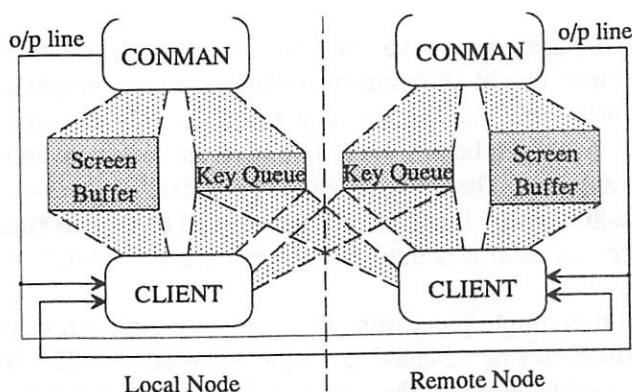


Figure 3 Centralized server scheme

An alternate suggestion was to share the key queue data units with clients on all nodes and to let each client update its local screen buffer. The kernel would propagate the keystroke information and free the clients from messaging overhead. Figure 4 shows the relation between tasks and data units in this third approach. Each client attaches each CONMAN's output line to an input so that it will be notified when a key is added to the queue. When a CONMAN pulses this output, the kernel passes the pulse to each client.

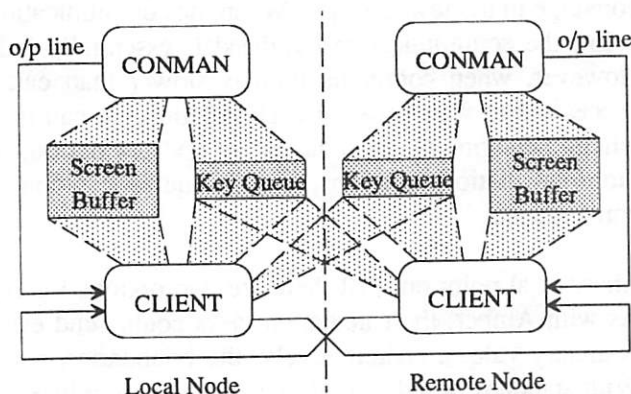
Unfortunately, even though CONMAN does not participate in passing its output pulses to the remote clients, it is blocked while this happens. Therefore, as Figure 2 shows, this approach, referred to as shared key queue, is slower than message passing. Also, the time necessary to send output line updates is proportional to the number of nodes. Therefore, the overall time grows linearly with the number of machines.

A simple modification of the third scheme yielded unexpectedly good performance and clearly indicated the value of asynchronous operation. The modification, as shown in Figure 5, was to have the local client pass the output pulse to remote clients, rather than depending on CONMAN. The thought was that this would free CONMAN to run in parallel with the output pulse propagation.



**Figure 4** Shared key queue approach

Figure 2 shows the dramatic performance improvement. It might have been expected that as the signal propagation burden increases with the number of nodes, the client task would have become a bottleneck. However, even for four remote nodes, this approach takes essentially the same time as the single node messaging scheme. What has happened is that while the local client waits for the output pulse to propagate, CONMAN continues to place keys in the key queue. Once the propagation is finished, the clients receive *all* of the keys in the queue and writes them to the screens. Thus, as the number of nodes increases, the keystrokes are grouped and network traffic per key to each node is reduced. This happens because we no longer require CONMAN and the clients to operate synchronously.



**Figure 5** Modified shared key queue

It is clear that the performance of each implementation of PHONE would be improved if support for multicasting were added to ARCADE. However, even with multicasting, the conclusion that asynchronous operation will significantly improve system performance is still valid.

#### 4. Cooperative Computation Example

The next example clearly demonstrates the difference between synchronous cooperation, supported by message passing, and asynchronous cooperation, using shared data. It involves solution of a boundary value problem by red/black successive overrelaxation. This is essentially the same example used by Chase et al to illustrate Amber[16] and it lends itself well to both solution approaches. Experimental results show that, as the number of processors increases, the

computational efficiencies offered by letting each machine run at its own pace outweigh the added communication overhead.

Red/black successive overrelaxation[17] is an iterative solution technique for finding the temperature at all points on a plate given the temperature at the edges. It uses LaPlace's equation which states that the value at a point equals the average of the values of the points surrounding it. A checkerboard grid is laid over the rectangle and the red and black squares are alternatively evaluated. The value at each red square is determined by its black neighbors and vice versa. Each iteration involves recomputing all of the red squares and then all of the black squares. The process continues until the values have converged.

With multiple processors, the solution region can be divided into rectangular subregions, each of which may be assigned to a separate processor. The bulk of the calculation involves points within the subregions, but the values at the boundaries must be communicated to the processors handling the adjoining subregions.

The frequency and method of this communication determine the speed of convergence and the complexity of the tasks. The Amber approach was to communicate the values of the red squares while the black squares were being calculated and vice versa. This allowed the algorithm to converge in the fewest steps. When the communication takes less time than the calculation of one color, the communication overhead is essentially hidden and attractive speed-ups are possible. However, when communication is slower than calculation, this synchronous approach forces processors to wait between each iteration. It can be shown [18] that an asynchronous method, where the subregion boundaries are updated periodically, also converges. Although it may take more calculation iterations, it will require less communication and can result in overall faster convergence.

Chase et al point out that there are two options for communicating the values at the boundaries. As with Amber, the calculation tasks could send each other messages containing the subregion boundary values. Alternatively, the boundaries, or even the entire rectangle, could be shared. With standard paged shared memory and an unlucky choice of page boundaries, this can lead to thrashing. However, with ARCADE, the shared area can be divided into data units and locking can be used to control updates and prevent thrashing.

Not surprisingly, the messaging approach leads to much more complex code. Each task must include the code necessary to transmit and receive the messages. With sharing, the kernel does the communication and the task just calculates. Also, explicit messaging requires the task to actively receive updates which makes asynchronous calculation difficult. The implicit communication used with data sharing allows that calculation task to ignore updates.

Two different ARCADE-based implementations of the red/black successive overrelaxation example were built and evaluated. Both operated on a relatively small rectangular region that was 160 x 100 points. The time to communicate between two nodes in the ARCADE interconnection was on the order of the time to calculate one fourth of the points. Thus, with more than two nodes, the communication overhead for the synchronous method dominated the calculation time. The evaluations consisted of running the calculation on a variable number of nodes and determining the time to convergence.

For both implementations, a calculator task was placed on each node and several other tasks supported the calculation. These other tasks provided an operator interface to configure the test,



a coordinator task to distribute the work and a graphical task to display progress. The first implementation followed the Amber approach and used messaging and synchronous processing. For a single node, it was able to converge in 29 seconds. When a second node was added, calculation time decreased, but only marginally, to 21 seconds. The communication overhead offset the calculation gain. For three nodes, the calculation time actually increased to 29 seconds. The added penalty of waiting for all three calculator tasks to synchronize every iteration simply swamped the computational savings. Performance continued to degrade as more processors were added. The speed-up curve in Figure 6 shows the dismal result.

The second implementation used distributed shared data and asynchronous calculation. The subregions were mapped onto data units and additional data units were used for the boundary regions. These boundary data units were shared between calculator tasks and were designed to minimize communication overhead and avoid multiple writer contention. Each calculation task dealt with two types of boundary regions. The *external boundaries* were the areas adjacent to its subregion. These provided the task with input data and were only read by the task. The *internal boundaries* were the areas on the edges of the task's subregion which were inputs to other calculator tasks. These the calculator task had to write.

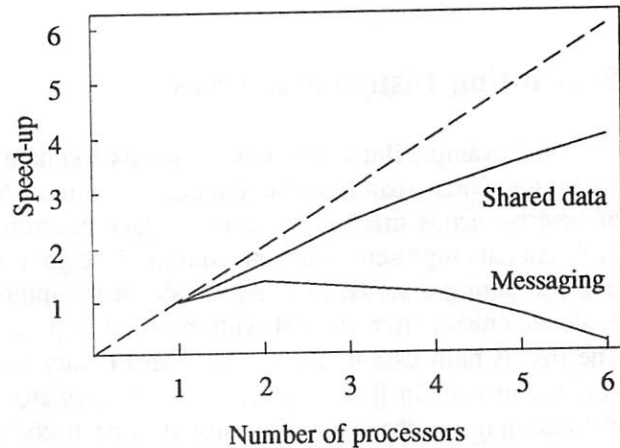


Figure 6 Speed-up curves for red/black

Figure 7 shows the subregion for a typical task. The heavier line indicates the actual subregion that this task has to calculate. The unshaded rectangles are the external boundary data units provided by other tasks and used by this task. The shaded data units are written by this task and used as external boundaries by other tasks. After a set number of iterations, the task copied the calculated values from its subregion into the shaded data units. It then obtained and released a write lock on these data units. This caused the kernel to propagate the new values to the replicas of these data units shared by other calculator tasks.

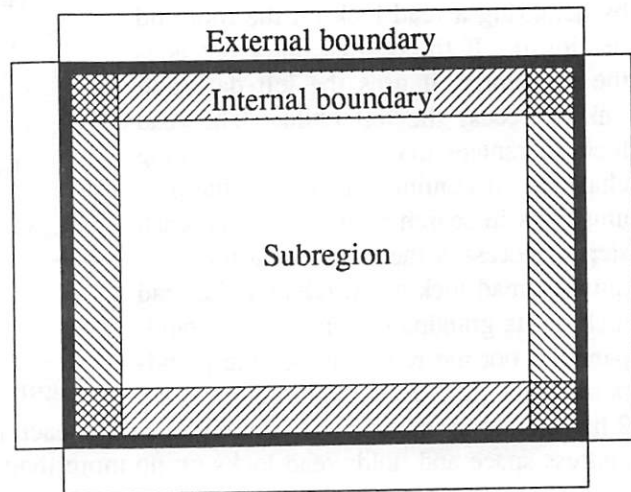


Figure 7 Calculator task data units

The single node time for this approach exceeded that for synchronous messaging. Using updates only every other iteration, it took 64 seconds to reach convergence. Part of this penalty is the extra overhead of copying data to the boundary data units. However, most of it is due to the additional iterations need for asynchronous evaluation. With two nodes, convergence time dropped to 37 seconds and for three



it was 26. Thus, for three nodes, the asynchronous, shared data method was better than the synchronous, messaging approach. As Figure 6 shows, it continued to improve, taking only 16 seconds for six nodes, the most tested.

The code need for the shared data calculator task was substantially smaller than that needed for messaging. For messaging, this task was about 1000 lines of C code, while for shared data, it was only 200. The major difference was that for shared data, there was no need to receive messages and determine what to do with them. This is essentially the same effect that was observed with PHONE.

## 5. Growing Distributed Trees

The final example illustrates how data units and data unit links can be used to create dynamic data structures which span machine boundaries. Since ARCADE can accommodate interconnections of heterogeneous machines, the techniques described here works even for machines which use different data representation conventions. The goal of the example is to create a distributed binary tree containing a sorted list. Any node in the interconnection should be able to look-up, insert or delete entries from the list with minimal impact on the same activity by any other node.

The tree is built essentially as a standard binary tree with each vertex being a data unit. Each vertex will contain five elements: the list entry stored at this vertex, data unit links to right and left children and the list entries stored at each child. It is necessary to store the children's list entries so that tasks at each node can properly manage locks on data units. Each task processing the tree shares the root data unit and never releases it.

Figure 8 shows a typical distributed tree being processed by two tasks. For simplicity, the list entries are just single letters. As noted, the root is shared by all tasks. When a task wants to look-up an entry, it begins by acquiring a read lock on the root and reading it. If the desired entry precedes the root entry, it uses the left data unit link to access the left child. The read lock guarantees that the link is not being changed. It continues to follow the data unit links in search of its entry. At each step, it accesses the target data unit, acquires a read lock on it, releases the read lock on its grandparent and, if the grandparent is not the root, releases the grandparent data unit. Thus, in the figure, Task 1 has looked-up the entry *U*. At any one time, each task has a maximum of four data units in its address space and holds read locks on no more than three.

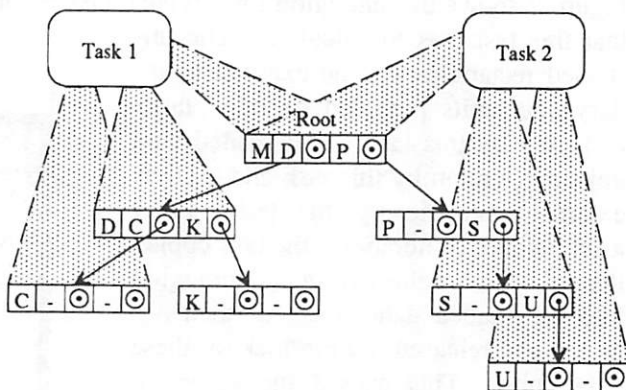


Figure 8 Distributed binary tree

Adding a vertex to the tree follows the same pattern. The task traverses the tree, starting at the root, until it finds a place for the new entry. It then releases its read lock on this last data unit and acquires a write lock. If some other task has a read lock on the same vertex, the task must wait. (Since ARCADE does not currently allow a task to upgrade a lock from read to write, it is conceivable that another task can sneak in and change the vertex. Therefore, the task must

reverify that this is the correct place to add its entry and that the entry has not already been added.) Once it has the write lock, it allocates a data unit for the new vertex and sets the data unit link in the parent to point at it.

Deleting an entry is only slightly more complex. Since ARCADE does not allow tasks to upgrade locks, the task must write lock the parent of the vertex to be deleted *before* it accesses the vertex itself. If it does not, the parent node could be changed between locks. Therefore, each vertex includes the list entries of its children. When deleting a vertex, all changes to the tree take place in the subtree that grows from its parent. Therefore, since the task has a write lock on the parent, it can eventually acquire whatever additional write locks it needs to properly restructure the tree.

With all of this locking and unlocking, it might seem that deadlocks are possible. However, there is a natural ordering of the data units in the tree and all locks are acquired in the same sequence. Therefore, deadlocks can not occur.

Since tasks have at most four data units in their address space at one time, even for very large trees, one may wonder where the other data units reside. As long as there is at least one data unit link pointing to a data unit, the ARCADE kernel maintains that data unit. In general, the kernel on a given node does not discard a data unit that is the target of a data unit on that node. When a data unit is released by one of the tree searching tasks, it is still being pointed at by its parent. Since that parent is still on this node, the data unit remains on this node. If it has to be accessed again later, no communication is necessary.

It is only by reaccessing that a task can be assured that it has valid data. If it kept a vertex in its address space and the vertex was deleted by another task, it would not know it. Since it acquires a read lock before using the data unit link in a vertex, it is assured that it has the latest version of the data.

The performance of the distributed tree was compared with that of a tree object, such as those of Orca. The tree object was located on one node, and tasks on other nodes sent requests to it. Early results indicate that the distributed tree takes approximately twice as long as the tree object to enter a given number of elements in the tree. This slowdown is primarily caused by the communication required to assert read locks in the current ARCADE implementation.

The preliminary test consisted of eight nodes which entered character strings into the tree. Each node entered a given number of strings as fast as it could and the total time was measured. Table 1 lists the total time for various tree sizes. Even though this test did not include deletions, it did indicate a problem with the distributed tree.

As noted above, in the distributed tree, each task read locks each node as it traverses the tree. Acquiring these locks requires communication, and the traversal is inherently slow. To verify that the read locks were the problem, a modified version of the distributed tree was developed. Read locking guarantees that the structure of the tree is not changed above the point where a task was adding an entry. Since the test did not involve deletions, it was possible to omit the read locks. It was still necessary to write lock the leaf where new data was added. This version actually outperformed the tree object. Its data is in the last column of Table 1.

Fortunately, there are implementations of read and write locks which allow read locking without communication. Such an implementation would likely allow the fully functional distributed tree to be competitive with the tree object.

## 6. Summary

We have distinguished between distributed shared memory, which relies on the paged memory model, and distributed shared data as realized in ARCADE, which uses data units. Although the three examples presented here used ARCADE, they also show how the more classic method can be helpful. The natural programming model provided in all examples would hold for any form of distributed shared memory. The speed up attributed to asynchronous operation is a direct result of the simplified model.

Size	Object	Distributed	No Lock
800	17	31	16
1600	35	66	30
2400	52	116	46
3200	69	150	61
4000	86	181	80
8000	188	376	156
16000	355	852	328

**Table 1** Time in seconds for 8 nodes to build tree

Two areas where the ARCADE approach has distinct advantages over paged memory are in locks and data unit links. Since locks give applications the ability to control updates, many of the inefficiencies encountered in traditional distributed shared memory are avoided. Data unit links offer a new paradigm for distributed shared data. It is not clear, for example, how to build a dynamic data structure like the distributed tree without a global pointer such as a data unit link.

## 7. References

1. Li, K., Shared Virtual Memory on Loosely Coupled Multiprocessors, Ph.D. Dissertation, Yale University, YALEU/DCS/RR-492, September, 1986.
2. Ramachandran, U., Ahamad, M. and Khalidi, M., Coherence of Distributed Shared Memory: Unifying Synchronization and Data Transfer, *Proceedings 1989 International Conference on Parallel Processing*, Volume II, August, 1989, pp. 160-169.
3. Ramachandran, U. and Khalidi, M., An Implementation of Distributed Shared Memory, *Workshop on Experiences with Distributed and Multiprocessor Systems*, USENIX, Oct., 1989, pp. 21-38.
4. Kessler, R. and Livny, M., An Analysis of Distributed Shared Memory Algorithms, *Proceedings 9th International Conference on Distributed Computing Systems*, IEEE, June, 1989, pp. 498-505.
5. Bal, H., Kaashoek, M. and Tanenbaum, A., A Distributed Implementation of the Shared

Data-Object Model, *Workshop on Experiences with Distributed and Multiprocessor Systems*, USNIX, Oct., 1989, pp. 1-20.

6. Cohn, D., Delaney, W. and Tracey, K., ARCADE - An Architecture for a Distributed Environment, Department of Electrical and Computer Engineering Technical Report 889, University of Notre Dame, Oct. 1988.
7. Delaney, W. The ARCADE Distributed Environment: Design, Implementation and Analysis, Ph.D. Dissertation, University of Notre Dame, April, 1989.
8. Cohn, D., Delaney, W. and Tracey, K., Structured Shared Memory Among Heterogeneous Machines in ARCADE, *Proceedings 1st Symposium Parallel and Distributed Processing*, IEEE, May, 1989, pp. 378-379, also Department of Electrical and Computer Engineering Technical Report 890, University of Notre Dame, Jan. 1989.
9. Tevanian, A. and Rashid, R., Mach: A Basis for Future Unix Development, Technical Report CMU-CS-87-139 Carnegie-Mellon University, June, 1987.
10. Acetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A. and Young, M., Mach: A New Kernel Foundation for UNIX Development, *Proceedings of Summer Usenix*, July, 1986.
11. Cheriton, D., The V Kernel: A Software Base for Distributed Systems, *IEEE Software*, April, 1984, pp. 19-42.
12. Armand, F., Gien, M. Herrmann, F. and Rozier, M., Revolution 89 or "Distributing UNIX Brings it Back to its Original Virtues", *Workshop on Experiences with Distributed and Multiprocessor Systems*, USENIX, Oct., 1989, pp. 153-174.
13. Bisiani, R. and Forin, A., Multilanguage Parallel Programming of Heterogeneous Machines, *IEEE Transactions on Computers*, Vol. 37, No. 8, Aug. 1988, pp. 930-945.
14. Cheriton, D., Problem-oriented Shared Memory: A Decentralized Approach to Distributed System Design, *Proceedings 6th International Conference on Distributed Computing Systems*, IEEE, May, 1986, pp. 190-197.
15. Tracey, K., *The Design and Implementation of an ARCADE-based Operating System*, Master's Thesis, University of Notre Dame, Notre Dame, IN, April, 1989.
16. Chase, J. et al., "The Amber System: Parallel Programming on a Network of Multiprocessors," in *Proceedings of the 12th Symposium on Operating System Principles*, ACM, November, 1989, pp. 147-158.
17. Ortega, J. and Voigt, R. Solution of partial differential equations on vector and parallel computers. *SIAM Review*, pages 149-240, 1985.
18. Bertsekas, D. and Tsitsiklis, J., *Parallel and Distributed Computation - Numerical Methods*, Prentice Hall, 1989.

1. ...
2. ...
3. ...
4. ...
5. ...
6. ...
7. ...
8. ...
9. ...
10. ...
11. ...
12. ...
13. ...
14. ...
15. ...
16. ...
17. ...
18. ...
19. ...
20. ...



# Virtual Memory Xinu

*Douglas Comer*  
comer@cs.purdue.edu

*James Griffioen*  
jng@cs.purdue.edu

Department of Computer Science  
Purdue University  
West Lafayette, IN  
47906

## ABSTRACT

The Virtual Memory Xinu Project investigates a new model of virtual memory in which dedicated, large-memory machines serve as backing store (page servers) for virtual memory systems operating on a set of (heterogeneous) clients. The dedicated page server allows sharing of the large physical memory resource and provides fast access to data.

This paper gives a brief overview of the Virtual Memory Xinu research project. It outlines the new virtual memory model used, the project's goals, a prototype design and implementation, and experimental results obtained from the prototype.

## 1. Background

Virtual memory operating systems provide mechanisms that allow programs requiring large amounts of memory to execute on a wide range of architectures regardless of the computer's physical memory size. This ability aids the development of portable code by allowing programmers to design and implement programs independent of the physical memory size available on the underlying machine. When user programs exhaust all available local physical memory, the operating system writes blocks of physical memory to the backing store. The virtual memory system later retrieves the blocks of memory from the backing store on demand.

Most conventional virtual memory operating systems use magnetic disks for backing storage<sup>8</sup>. Magnetic disks provide high data transfer rates, large storage capacity, and the ability to randomly access data, making them an appealing backing storage medium. The operating system usually reserves a fixed size region of the disk for backing storage and writes blocks of data directly to that region (i.e. no additional file structure or other organizational structure is imposed on the raw storage provided by the disk hardware)<sup>1, 5, 6</sup>.

More recent virtual memory systems have added a level of abstraction to the paging paradigm<sup>7, 10</sup>. These systems use the abstraction of files to hide the underlying storage device from the virtual memory system and allow the operating system to store data on the disk using high level file operations. The virtual memory system does not need to know the characteristics or

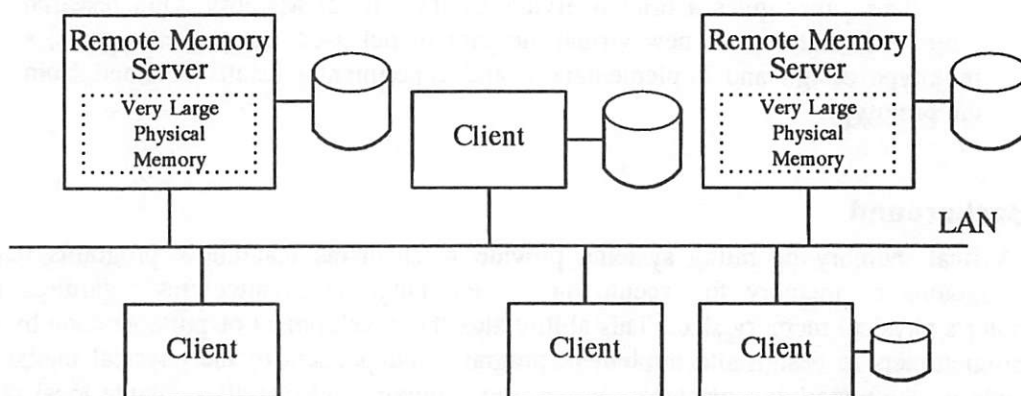
organization of the underlying disk device because the file system handles the storage of data on the disk. Some operating systems provide support for a distributed file system and allow diskless machines to use remote files for backing storage<sup>7,10,11</sup>. Unfortunately, using files for backing storage increases the overhead associated with paging. Writing data to a file usually requires a minimum of 2 disks accesses (1 or more to update the directory structure and 1 to write the data) whereas writing directly to the disk requires only 1 access<sup>1,5</sup>. Moreover, file systems often attempt to improve performance with techniques such as *read-ahead*. Read-ahead assumes that most programs access files sequentially and attempts to prefetch additional data whenever the system reads from a file. However, when applied to random access paging activity, prefetching wastes valuable buffer space, degrading both paging and file system performance.

The Virtual Memory Xinu project explores a new model of virtual memory in which dedicated, large-memory machines serve as backing storage for virtual memory systems operating on a set of (heterogeneous) clients. The dedicated memory server<sup>†</sup> allows sharing of the large physical memory resource and provides fast access to data.

This paper gives a brief description of the Virtual Memory Xinu project. It describes the model used, the system components that comprise the model, and their design.

## 2. The Remote Memory Model

The Virtual Memory Xinu project uses a new model for virtual memory called the *remote memory model*<sup>4</sup>. The remote memory model consists of several client machines, one or more dedicated machines called remote memory servers (or page servers), and a communication channel interconnecting all the machines. Figure 1 illustrates the architecture.



**Figure 1:** An Example Remote Memory Model Architecture

The model uses dedicated, large-memory machines called *remote memory servers* to provide backing storage for virtual memory systems executing on heterogeneous client machines. A virtual memory operating system executes on the client machines and provides the mechanisms needed to efficiently access the large-memory backing storage over a low delay, high bandwidth, communication channel. The large-memory page server machines provide backing storage for multiple heterogeneous client machines at speeds competitive with local disk speeds.

<sup>†</sup> We use the terms *memory server* and *page server* interchangeably.

Client machines share the large physical memory resource located at the page server. Each client machine has a private local memory large enough to support the normal processing demands. However, for jobs requiring large amounts of memory, clients page across the network, storing pages on the large-memory page server machine and retrieving them as needed. The page server allocates memory on demand to clients that require additional memory. Unlike other distributed virtual memory systems that page to a remote disk with fixed partitions, the remote memory model provides a large, dynamically allocated, shared, backing storage resource accessible to heterogeneous client architectures.

The goal of the Virtual Memory Xinu project is to design a highly efficient system in which heterogeneous client machines page across the network to shared, memory backing storage. In particular, the project investigates the design of a client virtual memory operating system with efficient support for remote memory backing storage, the design of a highly efficient memory server, and the design of a high performance reliable paging protocol.

### 3. System Design

#### 3.1. The Client Operating System

We assume each client machine provides hardware support for virtual memory and network communication. The operating system requires the hardware assist to support virtual address spaces and access remote memory backing storage. We envisioned a design in which the client virtual memory systems work closely with a page server to provide fast access to remote data. In addition, the client operating system must provide basic operating system functionality such as process management, process synchronization, interprocess communication, and device drivers. In particular, we identified the following design goals for the client operating system:

<b>Very Large Programs</b>	Large virtual address space support for programs requiring large amounts of memory.
<b>Multi-threaded Processes</b>	The ability to execute multiple threads of control within a process, allowing concurrent manipulation of shared data within a process.
<b>Multi-threaded Kernel</b>	The ability to execute several different tasks concurrently in the operating system kernel.
<b>Shared Memory</b>	Shared memory primitives that allow sharing of address-dependent data as well as providing an efficient mechanism for communication and synchronization between the threads in a process or between threads in separate processes.
<b>Hierarchical Design</b>	A design that arranges of the system components into a layered hierarchy resulting in a cleanly and elegantly designed system.
<b>Remote Paging</b>	Efficient access to a large, shared, memory resource for tasks requiring large amount of memory.
<b>Architecture Independence</b>	The ability to execute the client operating system on the wide variety of architectures composing the heterogeneous environment.

We designed the client operating system with these design goals in mind. We based the design on the Xinu operating system<sup>2,3</sup> because of its simplicity, its hierarchical design, and our familiarity with the system, allowing us to easily modify the system to meet our needs. Because Xinu did not support virtual memory, we were not constrained by an existing virtual memory design. Consequently, we were able to design a virtual memory system that exploits

the desirable properties of remote memory backing storage without sacrificing efficiency.

### 3.1.1. Background

Xinu<sup>2,3</sup> is a small, elegant operating system. It arranges the operating system components into a hierarchy of levels, clarifying the interaction between the various components of the system and making the system easier to understand and modify. Despite its small size, Xinu uses powerful primitives to provide the same functionality found in many larger operating systems.

Version 6 Xinu supplied primitives to handle memory management, process management, process coordination/synchronization, interprocess communication, real-time clock management, device drivers, and intermachine communication (a ring network). Version 7 Xinu replaced the point-to-point networking capabilities of Version 6 with support for the Ethernet and TCP/IP Internet protocol software. Version 7 also included a shell and a remote file system that allowed Xinu to access remote files via a remote file server executing on a UNIX system.

Both Version 6 and Version 7 Xinu ran all processes in a single, global, address space and did not use any virtual memory hardware. Before designing the VM Xinu operating system we examined several virtual memory hardware architectures and designed the system to accommodate them. In particular, the design was influenced by two substantially different architectures, the Vax<sup>†</sup> and the Sun<sup>‡</sup> architectures.

### 3.1.2. The VM Xinu Operating System

VM Xinu builds on the functionality provided by the small set of primitives found in Version 7 Xinu. The VM Xinu operating system executes on the set of client machines, providing virtual memory support for tasks requiring large amounts of memory and using remote memory for backing storage.

VM Xinu uses a hierarchical design to incorporate virtual memory support into the operating system, simplifying the kernel and clarifying the relationship between the various components of the system. The design also identifies and separates the architecture dependent components of the system from the architecture independent components. An architecture interface layer resides at the lowest level of the hierarchy, hiding the underlying hardware memory mapping support from the higher layers of the operating system and reducing the effort needed to port the system to new architectures.

VM Xinu uses the hardware's virtual memory support to create a large virtual address space for each application program. Each user process executes in its own address space and can only access data in that space. Each address space is defined by a mapping from virtual memory to physical memory and uses the MMU support to protect the data in the address space from all other user processes. When starting a user application, the operating system dynamically loads the user's program from the file system into a virtual address space and begins execution of the program. All dynamically loaded user processes execute in user-mode and trap into the kernel via system calls to invoke Xinu kernel routines. Because the kernel is protected from user processes, user processes only access kernel data structures via systems calls.

VM Xinu supports multi-threaded user processes allowing concurrent manipulation of shared data within a process. We define a thread as a point of execution within a process and its associated state information. All threads within a process execute instructions from the same text region, each at a different point in the code, and share the process' data region with all other threads in the process. Semaphores and other interprocess communication primitives

<sup>†</sup> Vax is a registered trademark of Digital Equipment Corporation

<sup>‡</sup> Sun is a registered trademark of Sun Microsystems Inc.



provide synchronization between threads, both between threads in the same process and threads in different processes. The system also provides the ability to execute multiple kernel tasks concurrently. Multiple lightweight kernel threads execute at different points in the kernel's text, carrying out kernel operations in parallel. Many operations performed by the kernel such as page reclamation, network management, and background paging are coded very simply and elegantly when viewed as concurrent kernel threads.

Many programming languages provide *address-dependent data structures* to support dynamically allocated data. Sorting and search algorithms often use address-dependent data structures such as linked lists or tree structures that have embedded pointers (virtual addresses). To accommodate sharing of address-dependent data structures, VM Xinu provides two types of data sharing. First, all threads executing within an address space share the data in the address space. Moreover, the data appears at the same location in each thread's view of the address space. Second, VM Xinu allows address-dependent data sharing between threads executing in different address spaces. The operating system reserves part of each address space, called the *shared/private* area. A thread can place address-dependent data in the shared/private area and then allow other threads to access it. Because of a novel design, the data always appears at the same location in all address spaces.

The VM Xinu operating system is unique in that it uses remote (physical) memory, accessed via a high speed, high bandwidth network, for shared backing storage. The kernel provides the virtual memory and networking mechanisms needed to page across the network to a remote memory server. Two kernel *dispatcher* threads handle the multiplexing and demultiplexing of all paging messages sent to and received from the memory server.

The kernel uses a highly efficient, reliable, data streaming, network architecture independent protocol to communicating with the memory server called the *remote memory communication protocol*<sup>4</sup>. The protocol consists of two layers: the Xinu Paging Protocol (XPP) layer and the Negative Acknowledgement Fragmentation Protocol (NAFP) layer. The XPP layer sends and receives high level paging messages while the NAFP layer handles the details of delivering XPP messages over the underlying communication channel. To insure end-to-end<sup>9</sup> reliable storage/retrieval of data, XPP uses timeouts and retransmissions. XPP supports data streaming using asynchronous message delivery, resulting in substantially higher throughput than synchronous protocols. To avoid overrunning the server with paging messages, the XPP protocol provides flow control mechanisms that limit the number of outstanding messages allowed at any given time.

The NAFP protocol fragments XPP messages into one or more packets and then reassembles the message at the destination. NAFP hides the details of the underlying communication channel from the XPP layer and provides transmission of arbitrarily large messages. NAFP attempts to correct fragmentation errors as soon as they occur using negative acknowledgements (NACKs). Early correction of errors using NACKs reduces the number of errors that reach the XPP level. NACKs improve efficiency in the presence of errors but contributes nothing to the overhead in the expected case, the case in which no errors occur. We refer the reader to [4] for a more detailed description of the communication protocol.

### 3.2. The Page Server

The page server provides memory backing storage for the virtual memory operating systems executing on the client machines. The design goals we used while designing the page server were:

**Heterogeneous Client Support** The server should provide remote memory backing storage to multiple heterogeneous clients simultaneously.



<b>Shared Resources</b>	Instead of preallocating fixed amounts of memory to each client, the memory resource must be shared between all the clients according to their needs.
<b>Arbitrarily Large Memory</b>	The memory resource provided by the server and presented to the clients must appear arbitrarily large.
<b>Shared Data</b>	The server should allow clients to share the data stored at the server with other clients.
<b>Fast Data Access</b>	To improve overall system performance, the server must efficiently locate and retrieve stored data.

The page server supports heterogeneous client architectures and operating systems regardless of their page size, word size, or byte order. The server uses the remote memory communication protocol to transfer pages of any size to or from the heterogeneous client machines, and dynamically allocates data structures to store the variable size memory segments clients send to the server. The server does not interpret the stored data; it simply returns the data in exactly the same form in which the data was received.

Instead of preallocating fixed amounts of memory to clients, all clients share the memory resource. When clients request storage space on the server, the server dynamically allocates regions of its memory to clients according to their needs. The only limit on a client is an indirect limit on the collective usage of the resource.

The page server uses a transparent two-level memory scheme to present an arbitrarily large memory resource to client machines. The server connects to one or more disk drives and uses a memory replacement policy to substantially enlarge the storage capacity of the server. From the client's viewpoint, the server simply provides a single level large memory resource.

To reduce the delay associated with retrieving memory from the remote memory server, the server attempts to minimize the time spent searching the data structures for the desired data. The server uses a double hashing algorithm to retrieve data in constant time. The data structures used by the server allow clients to share the data they store on the server with other clients. Not only does sharing allow different clients to access the same data, but it also reduces the memory used on the server by eliminating redundant copies of data (common program text, shared libraries, fonts, etc).

#### 4. Prototype Implementation

We designed and implemented a prototype distributed system based on the remote memory model. The system consists of heterogeneous client machines (Sun Microsystems SUN 3/50's, Digital Equipment Corporation Microvax I's and II's), a memory server, and a remote file server, all connected by a 10 Mb/sec Ethernet. The Sun and Microvax client machines simultaneously access the remote memory server for backing storage, demonstrating support for heterogeneous clients.

A *page server* provides backing storage for the VM Xinu kernel. Our prototype page server runs as a UNIX user level process, allowing us to run the server on a wide variety of platforms. We have used a SUN 3/50, Sparcstation, Vax 11/780, Microvax II and III, Vaxstation, Decstation, an 8 processor Sequent Symmetry, running a wide variety of operating systems (SunOS, 4.3BSD, Dynix, Ultrix, VM Xinu) as the page server machine. We built the high speed communication protocol used to reliably transfer pages between the client and the page server on top of UDP, allowing clients to access page servers on other networks across one or more gateways. The combination of a highly efficient memory server and a special purpose communication protocol results in a high-speed, shared, secondary storage mechanism operating at speeds competitive with a local disk.

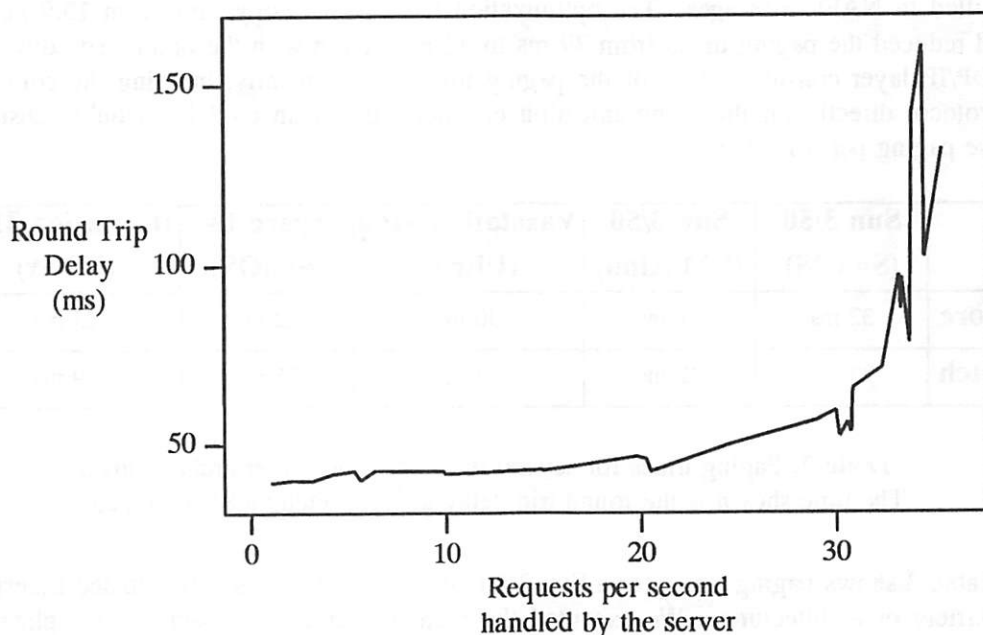
## 5. Experimental Results

Initial performance results, obtained from diskless Sun 3/50 client machines paging across a 10 Mb/sec Ethernet to a Sun 3/50 memory server, are described in [4] and show a significant improvement over diskless systems that page to a remote file system (see table 1). The results show that the time to service a page fault in SunOS takes twice as long as the time to service a page fault in VM Xinu. In addition, SunOS pays a high cost to insure that each store operation commits the data to disk before acknowledging the operation. As a result, VM Xinu page store times are 4 times faster than SunOS. In VM Xinu, unlike SunOS, the round trip delay for a write request and the round trip delay for a read request are symmetrical.

	VM Xinu	SunOS
Random Read	39 ms	84 ms
Random Write	39 ms	176 ms

**Table 1:** Paging access times for VM Xinu and SunOS 4.0

Figure 2 illustrates the memory server response time for various server loads. We used a Sun 3/50 as the memory server with Sun 3/50 machines running VM Xinu as client machines. We generated the various server loads by varying the number of clients and the number of requests per second issued by each client. At 30 requests per second the prototype page server becomes saturated and any additional load on the server significantly increases the round trip delay. However, for loads of less than 30 request per second the results show that the average round trip delay for store and fetch requests remains less than 56 ms, regardless of the number of clients. As long as the combined request rate of all the client machines remains less than the saturation rate, we can add new clients without degrading the performance of existing clients.



**Figure 2:** Remote memory delay for various server loads

The paging times reported in [4] measure the combined performance of the remote memory server and the communication protocol. Our experience with the system indicated that the communication protocol, rather than the memory server, consumed the majority of the paging time. To improve the paging performance, we focused our attention on the communication protocol in an attempt to streamline the protocol and reduce the communication overhead.

Component	Time	Percentage
XPP/NAFP	8.6 ms	22%
UDP/IP	23.6 ms	61%
Memory Server	6.8 ms	17%

**Table 2:** Breakdown of the time required to process a paging request.  
The percentage is calculated from a total paging time of 39 ms.

Because NAFP hides the underlying communication channel from the XPP protocol, we can run the communication protocol over most network architectures. In our prototype implementation, we built the communication protocol on top of the UDP protocol, using the virtual network provided by the IP protocol to allow access to memory servers on remote networks. Because our design does not bind the remote memory communication protocol to a specific type of communication channel, we wanted to measure the additional overhead incurred as a result of our implementation decision to use UDP as the communication channel.

Table 2 shows the breakdown of a paging request in terms of the time spent processing each stage of the request. The table shows that the majority of the paging time can be attributed to the UDP/IP protocol (over 60%). Upon closer inspection, we observed that the UDP/IP layer spent 59% (13.9 ms) of its time copying data (i.e. the contents of the page being stored/fetched). Consequently, we optimized the UDP/IP copy routine to efficiently copy the large blocks of data transmitted in NAFP messages. The optimization reduced the copy time from 13.9 ms to 4.0 ms and reduced the paging times from 39 ms to 32 ms. Even with the optimized copy routine, the UDP/IP layer consumes 42% of the paging time. Consequently, building the communication protocol directly on the communication channel rather than on UDP would substantially improve paging performance.

	Sun 3/50 (SunOS)	Sun 3/50 (VM Xinu)	Vaxstation 3100 (Ultrix)	Sparc 1+ (SunOS)	Decstation 3100 (Ultrix)
<b>Store</b>	32 ms	31 ms	30 ms	22 ms	23 ms
<b>Fetch</b>	32 ms	32 ms	34 ms	25 ms	29 ms

**Table 3:** Paging times for several different page server architectures.  
The time shown is the round trip delay to store/fetch an 8K byte page.

Table 3 shows paging times for a Sun 3/50 client paging across a 10 Mb/sec Ethernet to a wide variety of architectures. We executed the memory server as a user level application on each architecture and measured its performance. Each column of the table shows the time required to store/fetch an 8K byte page to/from the specified architecture/operating system. The

times show that the architecture and the operating system both have a significant impact on the paging times. Clearly, the faster machines (the Decstation 3100 and Sparc 1+ RISC architectures) outperform the slower architectures (the Sun 3 and Vaxstation). In particular, the RISC machines provide backing storage at speeds competitive with local disks. Moreover, because our implementation builds on the UDP/IP protocols, the operating system's implementation of these protocols has a significant effect on paging times. SunOS sends and receives packets at roughly the same speed, while Ultrix sends packets substantially slower than it receives packets. Although the optimized copy routine reduces the time VM Xinu spends processing a message by 9.9 ms, we only see a 7 ms speed-up over the old Sun 3/50 page server times (table 1). In this case, SunOS is the bottleneck. The lower bound on paging times (32 ms) is based on the rate at which SunOS can process UDP/IP packets.

In short, the results show that our implementation decision to use UDP/IP as the underlying communication channel significantly impacts the system's performance. However, even with the extra overhead resulting from UDP/IP, the system performs at speeds competitive with a local disk. The prototype clearly demonstrates the viability of building hierarchically-layered systems paging to remote memory backing storage without sacrificing efficiency.

## 6. Conclusions

Using the remote memory model as an alternative model for designing distributed systems has many attractive properties. The large memory resource shared by all client machines is especially appealing. Experience with the prototype system clearly demonstrates the viability of the remote memory model and shows that performance is competitive with distributed systems currently in use. Finally, we showed that the remote memory model can support heterogeneous clients machines without sacrificing efficiency.

## 7. Prototype Availability

An implementation of the VM Xinu system, including source code, for the Sun 3 architecture is now available and can be ordered from The Xinu Project, Computer Science Department, Purdue University, West Lafayette, IN 47907, or via email from [xinu-info-request@purdue.edu](mailto:xinu-info-request@purdue.edu). The memory server runs as a user level process on most Unix systems. The distribution tape includes source code for the VM Xinu kernel, the Xinu remote file server, the memory server, a UNIX simulation library, and several VM Xinu application programs.

## 8. Acknowledgements

We would like to thank Guoben Li for his contributions to the implementation and experimental evaluation of the prototype.

## References

1. Maurice J. Bach, *The Design Of The Unix Operating System*, Prentice Hall, 1986.
2. Douglas Comer, *Operating System Design: The Xinu Approach*, Prentice-Hall, 1984.
3. Douglas Comer, *Operating System Design, Volume II: Internetworking with Xinu*, Prentice-Hall, 1987.
4. Douglas Comer and James Griffioen, "A New Design for Distributed Systems: The Remote Memory Model," Proceeding of the Summer Usenix Conference, June 1990.
5. Samuel J. Leffler, Marshal K. McKusick, Michael J. Karels, and John S. Quarterman, *The Design and Implementation of the 4.3BSD Unix Operating System*, Addison Wesley, 1989.

6. Henry Levy and Peter Lipman, "Virtual Memory Management in the VAX/VMS Operating System," *Computer*, pp. 35-41, March 1982.
7. Michael N. Nelson, "Virtual Memory for the Sprite Operating System," Tech Report UCB/CSD 83/301n, University of California Berkeley, June 1986.
8. James L. Peterson and Abraham Silberschatz, *Operating System Concepts*, Addison Wesley, 1985.
9. J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-To-End Arguments in System Design," *ACM Transactions on Computer Systems*, vol. 2, pp. 277-288, 1984.
10. Robert A. Gingell and Joseph P. Moran and William A. Shannon, *Virtual Memory Architecture in SunOS*, Sun Microsystems, Inc., 1988.
11. Brent B. Welch, "The Sprite Remote Procedure Call System," Tech Report UCB/CSD 86/302, University of California Berkeley, June 1986.



# Early Experiences with the GOTHIC Distributed Operating System\*

Isabelle Puaut Michel Banâtre Jean-Paul Routeau

IRISA-INRIA

Campus universitaire de Beaulieu,  
35042 Rennes Cedex (France)

e-mail : puaut/banatre/routeau@irisa.fr

## Abstract

This paper is devoted to an early evaluation of the design and use of the GOTHIC distributed operating system, built at IRISA/INRIA (France). The design of GOTHIC started in 1986 and a first prototype is available since June 1990. The salient features of GOTHIC include a network-wide memory management and the use of a fast stable storage to build reliable applications. GOTHIC provides the abstraction of a shared memory on a distributed architecture, the shared space being a segmented single level store. Reliability is achieved by the addition of fast stable storage devices. This paper describes in details the structure and performance of GOTHIC, and presents some of the most significant experiences to date.

**Key Words and Phrases:** distributed operating system, fault tolerance, stable storage, virtual memory management, parallelism

## 1 Introduction

A distributed system is potentially more powerful than a centralized one, in the following ways: a distributed system can be more reliable, because some of its functions can be replicated; it can be more efficient, because computations can be carried out in a parallel way; finally, it can be extended by adding processing or storage elements. Ideally, these advantages should not make the system more difficult to use. Let us take the definition of the ideal distributed operating system given by Tanenbaum in [TR85]:

*A distributed operating system is one that looks to its users like an ordinary centralized operating system, but runs on multiple, independent CPUs. The key concept here is transparency, in other words, the use of multiple processors should*

---

\*Gothic is an INRIA/BULL project

*be invisible (transparent) to the user. Another way of expressing the same idea is to say that the user views the system as a "virtual uniprocessor," not as a collection of distinct machines.*

The key aim in the design and development of GOTHIC, which is built on top of a local area network of loosely coupled multiprocessors is *transparency*. Transparency is provided both at the programming language level and at the operating system level.

At the programming language level, we have designed a new parallel programming language called POLYGOTH [Ben90]. In POLYGOTH, applications are structured into *objects*. The language integrates two fundamental notions : the *multiprocedure* construct to express parallelism and the notion of *object fragment*. The multiprocedure is an extension of the procedure concept integrating procedural control and parallelism [BBP86]. In POLYGOTH, an object can be built out of several fragments, possibly located on different processors. Multiprocedures are defined as access methods to fragmented objects. Multiprocedures and fragments allow to exploit the parallelism of the distributed architecture without knowledge of the physical location of the parallel components and the data they manipulate. In other words, transparency to distribution is provided at the programming language level. As this paper is focusing on the operating system aspects, we do not give more details about the POLYGOTH language here.

At the operating system level, transparency is achieved in several ways. In this paper we will concentrate on two main aspects : fault tolerance and memory management. Fault tolerant mechanisms allows masking of processor crashes, and therefore achieve fault transparency to users. Section 2 describes a hardware device, the *stable storage board* (SSB), used to realize fault tolerant services. The experience gained with the design and the use of this stable storage board is also given in section 2. Section 3 describes the memory management principles implemented to achieve transparent access to data, and report some lessons learned by using this memory management. Some conclusions and directions for further research are given in section 4.

## 2 Fault Tolerance

### 2.1 Basic idea

Hardware redundancy may be coupled with software techniques, so as to provide fault tolerance [Bar81, BBG83]. Our idea is to experiment the use of a fast stable storage (hardware) device to build cheap fault tolerant machines based on data redundancy.

A stable storage is a reliable device with the following properties:

1. *Resistance against external failures*: a piece of data stored in stable storage is resistant both to hardware failures (e. g. processor or power failures), and to software failures (those leading to a bad memory access).

2. *Atomicity of read and write accesses*: an operation on an object stored in stable storage either completes successfully, or fails. In the latter case, all changes made on the object are undone so as to restore the initial state of the object.

Existing solutions to design stable storage devices are based on data redundancy. The most popular stable storage implementation has been proposed to build a reliable distributed file system [Lam81]. In order to provide resistance against disk failures and decays, each file block is replicated on two independent drives. This solution is acceptable for file servers, since the grain of recovery is rather coarse. However, when dealing with small objects or when a fine grain of recovery is needed, a disk implementation of stable storage is inefficient due to disk access time.

As far as GOTHIC is concerned, the following characteristics of the stable storage were required. First, the access time of the stable storage should be comparable to the access time of a RAM memory. Second, it should be possible to handle atomically complex data structures. Finally, the stable storage should be tolerant to processor malfunctioning.

In order to achieve these objectives, we have designed a fast stable storage board (SSB) [BBM88]. We now describe the stable storage board into more details. Then, we report experience gained both in designing and using the stable storage board.

## 2.2 The stable storage board

### Hardware structure

In order to implement the stability property, there should be at least one valid copy of any object at any time. Every object stored in stable storage is composed of two copies located on two independent memory banks. Accesses to these banks are mutually exclusive. Note that the two banks memory allows the detection of memory decays and their correction assuming that a valid copy of the object exists (on one of both banks) and can be identified.

High performance is guaranteed by building the stable storage board from non-volatile RAM memory banks belonging to the processor address space. Access control is ensured by internal mechanisms which are simple enough to be implemented in hardware.

Complex data structures are handled by means of atomic update of a group of objects.

Stable storage tolerance to processor malfunctioning implies *auto-protection* of stable storage against processor crashes and *autonomy*. Auto-protection is achieved through very strict access control. Autonomy means that the stable storage is able to take some decisions. For example, the stable storage is able to detect that the processor (accessing it) is faulty, and then initiates a reconfiguration. Further information can be found in [BBM88].

### Integration of the stable storage board into a multiprocessor machine

Integrating a stable storage board into our architecture is rather straightforward. Every stable storage board may be accessed via the local bus of the processor it is attached to, or

via the global bus of the multiprocessor it belongs to, as depicted in figure 1.

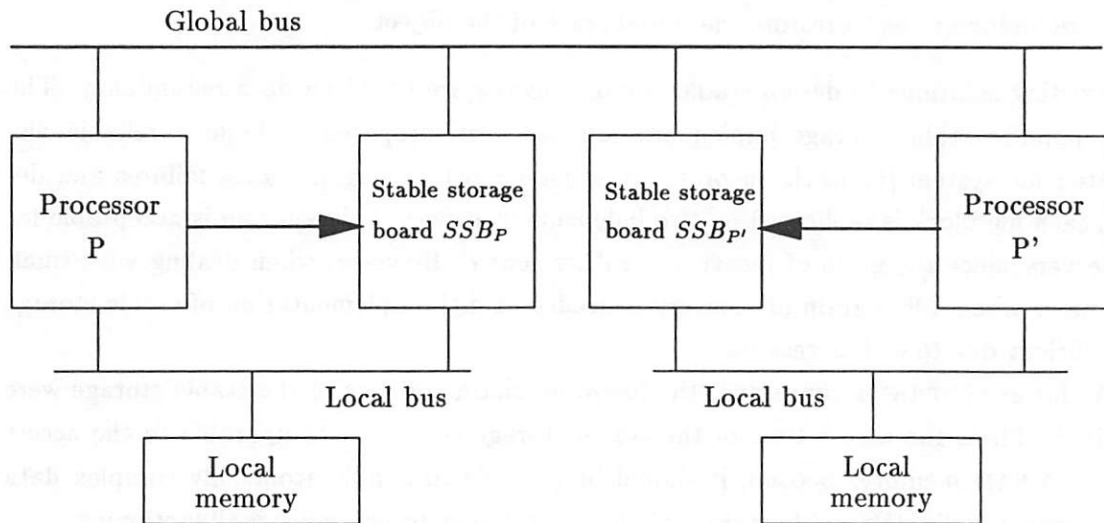


Figure 1: Architecture of a fault tolerant multiprocessor.

An interesting feature of this architecture is the ability to provide a normal service, even if some processing unit fails. Actually, it is sufficient that one processing unit (of the multiprocessor) survive and that the global bus service be available in order to get a normal service, as explained hereafter.

Let us consider process  $p$  running on processor  $P$  equipped with a stable storage board  $SSB_P$ . From time to time  $p$  may store its state into  $SSB_P$ , so as to establish a *checkpoint* for example. The detection of a crash of processor  $P$  is performed by the stable storage board *itself*. Every fixed time period,  $SSB_P$  sends a signal to  $P$ . If this signal is not acknowledged by  $P$ , then  $SSB_P$  assumes that  $P$  has crashed. It is then necessary to make the stable information accessible by another processor. This is done by granting access to another processor  $P'$  belonging to the same multiprocessor. In normal use,  $SSB_P$  is accessed by  $P$  through its local bus. When  $P$  fails, it is accessed by another processor  $P'$  through the machine global bus. These two access links are mutually exclusive.

### Performance of accesses to stable storage

Performances of the stable storage board can be illustrated by comparing them to those of a RAM memory (Table 1).

One can see that the access time of the stable storage board ranges between two and three times the access time of an ordinary RAM memory. Accessing SSB being comparable to main memory access time, the SSB can be used to store process checkpoints efficiently.

Update Operation	SSB	RAM
Byte	12.6 $\mu$ s	3 $\mu$ s
Word (16 bits)	12.6 $\mu$ s	3 $\mu$ s
Word (32 bits)	17.4 $\mu$ s	5.4 $\mu$ s
1024 bytes	1.9ms	0.9ms

Table 1: Comparison of performances between a RAM memory and the SSB

### 2.3 Fault tolerance management in the kernel

The functions of the stable storage board have been integrated into the GOTHIC kernel in two layers.

The lower layer deals with stable storage physical management and provides the operations to create and delete objects (single objects as well as groups of objects). This layer also manages stable storage initialization, processes the interrupts raised by the stable storage board.

Higher-level services can be built on top of this level. A checkpointing service allows atomic saving of process states (stack and data) in stable storage. Such processes are called *stable processes*. After processor crash, the checkpointing service restarts aborted process from their last state which is stored in stable storage.

These facilities have been used to build fault tolerant applications. For instance, a reliable communication system [Mor90] has been built. This communication system provides reliable point to point message transmission and atomic multicast even in presence of node crashes and communication failures.

### 2.4 Lessons related to fault tolerance

Experience gained both in designing the fault tolerant multiprocessor and in programming reliable applications is reported below.

#### Using a stable storage board to build a fault tolerant machine

The extension of the multiprocessor machine to a fault tolerant one was rather an easy task. The initial multiprocessor hardware was not modified. Stable storage boards were designed and just added to each processor. This same approach can be followed to build cheap general purpose fault tolerant machines from standard architecture, and is being experimented by us within the framework of another project, the FTM (as Fault Tolerant Multiprocessor) project [BMRS91].

#### Advantages of recovery implemented at the hardware level

We believe that much benefit is gained when crash recovery services are performed by the stable storage logic itself independently of the processor. Should a processor crash, all the



objects are restored in their initial state without processor intervention. Note that this characteristic distinguishes our approach from related ones, such as SEQUOIA [Ber88]. In SEQUOIA a stable memory is also used (each piece of information being replicated in two distinct memory blocks). However, unlike GOTHIC, crash recovery has to be performed entirely by the processor itself.

### Usefulness of atomic group update

As stated before, the stable storage board allows atomic handling of complex data structures by means of an atomic group update operation. This operation ensures that either all objects of a group are updated or none of them if a malfunction occurs. The design of the reliable communication system described in [Mor90] showed the usefulness of this operation. When sending a message, the message contents together with its sequence number are stored in stable storage. If a crash occurs, the communication system is then capable to retrieve the (presumed) unaltered message from stable storage so as to perform retransmission. The message contents and its sequence number are separate objects, which must be stored atomically. This is achieved in a straightforward manner by the atomic group update operation offered by the stable storage board. Note that without this facility, the application programmer would have the burden to manage by himself correct programming of the atomicity property on the group of objects.

## 3 Memory management

### 3.1 Principles

Programming with shared data is well understood. Even in non-shared memory architectures, many programming languages based on logical shared memory have been proposed [BF88, JLHB88]. GOTHIC offers the abstraction of a shared memory on a distributed architecture on top of the message passing facility provided by the system. This logical shared memory is referred to as *distributed shared memory* (DSM) in the following. In GOTHIC, processes can be involved in independent computations, that require modularity and protection. They share a segmented single level store [Mic90, Roc90], composed of a set of *segments*. We now describe into more details the way processes access segments and the protocol used to maintain segments consistency. Then, some implementation details are given. Finally, we report the experience gained both in building and using the GOTHIC DSM.

#### Accessing a segment

A *segment* is a unit of permanent information, directly addressable according to the single level store principle. Each segment is represented on one disk, and can migrate. A segment

is internally named using a location independent Universal Unique Identifier (UUID). This UUID gives hints for the segment location.

A segment can be dynamically *mapped* or *unmapped* into a range of virtual addresses of a process. Each segment is divided into pages, brought into main memory on demand. Main memory retains the most recently used pages of segments. Since the segments backing stores are spread over the network, remote paging can occur.

Segment location is transparent to the user. Accessing a segment only requires the knowledge of its UUID. Thus, the user is completely unaware of the physical location and distribution of the segments.

### Sharing a segment

As the main purpose of the GOTHIC DSM is to provide a convenient tool to implement inter process communication based on shared data, a strong consistency semantics, as defined in [CF78] is enforced : *a memory scheme is consistent if the value returned by a LOAD instruction is always the value given by the latest STORE instruction with the same address.* Rephrasing this property in the GOTHIC environment means that the value returned by a read instruction of a word within a segment is always the last value written (read and write instructions may be executed by different processes residing on distinct machines).

The coherence protocol designed to enforce this semantics [Roc90] is the Berkeley protocol [KEW<sup>+</sup>85] which we have modified so as to take into account the distributed environment of GOTHIC.

## 3.2 Implementation

The GOTHIC kernel was obtained by rewriting an existing kernel. Memory management was integrated into this kernel in three steps. The first step consisted in providing a paged virtual memory management and a single level store on a single processor machine. This step has required a complete redesign of the initial kernel memory management: page fault handlers have been rewritten, the whole physical memory management has been changed, and a permanent information management system has been built. The second step was the extension of this kernel to a multiprocessor, the implementation of the consistency protocol being the main work. The third step dealt with the extension of the kernel to a local area network of multiprocessor workstations. In the following, we mainly focus on the implementation of the consistency protocol.

Our machine deals with segments using a linearly structured *page table*, locating the page frames of that segment in main memory. On a given processor, the processes sharing the same segment also share its page table, so that each process refers to the same resident page frames. When a segment is mapped into a process address space, an entry in the process segment table is created and linked to that segment page table. Pages are 1K-bytes in the current implementation.

The Berkeley protocol we have extended was initially designed to maintain consistency of memory blocks in tightly coupled (shared memory) multiprocessors with multiple hardware caches. A memory block may be replicated in different hardware caches. Consistency is enforced by allowing any number of caches to read a given block, but allowing only one cache at a time to write the block ; an attempt to write into a replicated block (consistency violation) results in the invalidation of its replicas. We have been following a similar protocol in GOTHIC : shared memory in the Berkeley protocol should be read as segment in the GOTHIC environment ; memory block should be read as segment page ; hardware cache should be read as local memory. Detection of coherence violations are achieved by using the virtual memory address translation mechanism in the following way.

Pages whose copies are not present in memory are marked as invalid in the page table in order to trigger a hardware interrupt. Replicated pages are marked read-only in the segment page table, while non-replicated pages are marked read-write. An attempt to write into a read-only page results in the invalidation of its replicas. The current location of replicated pages is memorized to avoid broadcasts of invalidation messages. A consistency server is associated with each processor. The server memorizes the access rights and current location of the pages belonging to segments present on its disk. Given these pieces of information, the server can resolve page faults. Reliable message passing is used to communicate with the consistency server.

### 3.3 Some performance measures

The basic performances of memory management have been evaluated by measuring the time necessary to handle read page faults. These measurements have been performed on an architecture consisting of two multiprocessor machines, connected through a moderately loaded ETHERNET network. The figures obtained by the measures are summarized in table 2.

	Resident in main memory	Resident in disk
Local to the machine	14ms	35ms
Remote to the machine	270ms	300ms

Table 2: Timing measurement on a read page fault

Several cases have to be considered. First, the page might be local to the machine or residing on a remote machine. Second, on a machine, a page might reside in main memory (of one of the processors belonging to the machine) or in secondary memory (disk).

These figures show that the time necessary to handle a remote page fault is approximately ten times more than to handle a local one. This somewhat low performance when handling remote page faults is in our view partly due to the relatively poor performances of the low level communication system.

Note that in the different cases of local page faults, the times necessary to handle the fault are close to each other.

### **3.4 Lessons related to memory management**

Here are described both positive and negative aspects of programming upon our distributed shared memory system. Some of the design issues are also evaluated.

#### **Development of distributed applications using the Gothic DSM**

Recall that the DSM developed in GOTHIC provides location transparency, easy sharing, and cacheing. These three features greatly influenced the applications built on GOTHIC. We illustrate this in the following.

The distributed implementation of the parallel object-oriented language POLYGOTH (see section 1) makes intensive uses of the DSM properties to simplify the language run time support. Objects (and object fragments) of POLYGOTH program are stored in segments. Parallel components of multiprocedures are executed on distinct processor so as to speed up the program execution. These parallel components access the shared segments. Consistency of objects is directly insured by the strong consistency semantics of the DSM and no further work is needed.

GOTHIC DSM has been also used to implement an image processing application: a parallel ray tracer [BP90]. Given a scene description, this program computes the image as seen by an observer. Both the scene description and the image are stored in shared segments. The segment containing the image is divided into areas. The computation is achieved by a collection of processes, residing on different processors ; each of them computes an area, and once it is finished, selects another one and computes it. The most recently used pages of the input segment containing the scene description (which can be very large), are retained in main memory without any intervention of the ray tracing program ; moreover, the output segment containing the image is kept consistent in a transparent way.

These applications were straightforward to develop, and convinced us that the principles of the GOTHIC distributed shared memory are sound and effective.

#### **Performance aspects**

The GOTHIC DSM provides a uniform way to access segments independently of their location, these being mapped on demand into the process virtual address space. Many factors may impinge performances. We comment two main factors in the following.

- In centralized operating systems, it is known that applications using virtual memory management must possess a good locality of reference in order to obtain good response time. This property is still more important when using DSM since page fault handling may require communications with remote nodes (recall that in our implementation, a remote page fault is ten times more expensive than a local one).
- Intensive data sharing can lead to surprisingly bad results. For example, assume that multiple producer processes residing on different processors and running in parallel produce information items into a shared buffer represented as a single segment page. The strong consistency semantics of the DSM may induce a lot of page faults, the page being exchanged forth and back between the set of producers. This effect is known as the “ping-pong” effect [DSB88].

In summary, data within segments should be carefully organized so as to optimize performances.

### Modularity

A potential drawback in our management of consistency comes from the fact that it is realized at the kernel level. Unlike MACH [ABB<sup>+</sup>86] and CHORUS [RAA<sup>+</sup>88], a user can not connect external pagers to the kernel. In other words, user defined paging policies can not be built. This implies that only one (default) paging policy exists. Memory management is therefore less extensible and modular than in MACH or CHORUS microkernels. However, despite this lack of modularity, one advantage of using a system defined policy is that the overhead induced to call user defined paging policies is avoided.

### Fault tolerance

A remaining problem to be solved in our DSM is processor fault handling. When a process crash occurs, the data recently modified by this processor is lost. Furthermore, some segments can be inconsistent if some of their pages have been swapped out to disk and others have not. Maintaining segment integrity in presence of processor crashes and sharing is an issue to be taken into account. This is further discussed in the last section.

## 4 Conclusions

In this paper, we have described the main features of the GOTHIC distributed operating system, and we have drawn some observations about its design and use.

Experience with programming on the GOTHIC system has shown that the implementation of reliable applications can take advantage of the built-in atomic transaction features of the stable storage board. In our view, the most significant success of GOTHIC concerns fault tolerance. We have come to the conclusion that it is possible to design cheap fault tolerant



machines from standard open architectures by means of a fast stable storage. Concerning the distribution aspect, we are convinced that a distributed shared memory is a powerful tool to build parallel and distributed applications.

These two features of GOTHIC have been studied independently. As stated in 3.4, the GOTHIC virtual memory management system does not take into account processor crashes. One could imagine using a fast stable storage to build a fault tolerant memory management system, so that memory pages would be swapped out to stable storage (copying from stable storage to secondary storage being done in the background). To solve this problem, we are currently working on the design of a *transactional memory* [BJ90]. Each update of a memory page is included into an *atomic action* managed by the transactional memory. Our current research also covers the design of a fault tolerant kernel based on the MACH kernel, using a stable transactional memory [BMRS91, BMRS90].

## Acknowledgements

We would like to thank J. P. Banâtre, T. Leconte, P. Lecler, B. Michel, C. Morin, G. Muller, F. Poyette and B. Rochat for their contribution to the design and implementation of GOTHIC. We also like to thank V. Issarny, M. Jegado, G. Lapalme and D. Le Metayer for their helpful comments on earlier drafts of this paper.

## References

- [ABB<sup>+</sup>86] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for unix development. In *Proc. of Usenix 1986 Summer Conference*, pages 93–111, June 1986.
- [Bar81] J. Bartlett. A nonstop kernel. In *Proc. of 8th ACM Symposium on Operating Systems Principles*, Pacific Grove, December 1981.
- [BBG83] A. Borg, J. Baumach, and S. Glazer. A message system supporting fault-tolerance. In *Proc. of 9th ACM Symposium on Operating Systems Principles*, pages 90–99, Bretton Woods, N.H., October 1983.
- [BBM88] J. P. Banâtre, M. Banâtre, and G. Muller. Ensuring data security and integrity with a fast stable storage. In *Proc. of 4th International Conference on Data Engineering*, pages 285–293, Los Angeles, February 1988. IEEE.
- [BBP86] J.P. Banâtre, M. Banâtre, and Fl. Poyette. The concept of multi-fonction : a general structuring tool for distributed operating systems. In *Proc. of 6th International Symposium on the Principles of Distributed Computing*, pages 478–485, Cambridge, May 1986. IEEE.
- [Ben90] M. Benveniste. Operational semantics of a distributed object-oriented language and its Z formal specification. Technical Report 1230, INRIA, May 1990.

- [Ber88] Ph. A. Bernstein. Sequoia: A fault-tolerant tightly coupled multiprocessor for transaction processing. *IEEE Computer*, pages 37–45, February 1988.
- [BF88] R. Bisiani and A. Forin. Multilanguage parallel programming of heterogeneous machines. *IEEE Transactions on Computers*, 37(8):930–945, August 1988.
- [BJ90] M. Banâtre and P. Joubert. Cache management in a tightly coupled fault tolerant multiprocessor. In *Proc. of 20th International Symposium on Fault-Tolerant Computing Systems*, Newcastle, June 1990.
- [BMRS90] M. Banâtre, G. Muller, B. Rochat, and P. Sanchez. A reliable distributed virtual memory on top of the mach kernel. In *OSF Micro Kernel Applications Workshop*, Grenoble, France, November 1990.
- [BMRS91] M. Banâtre, G. Muller, B. Rochat, and P. Sanchez. Design decisions for the ftm : a general purpose fault tolerant machine. Technical report, INRIA, Rennes (France), 1991.
- [BP90] D. Badouel and T. Priol. An efficient ray tracing algorithm on a distributed memory parallel computer. Technical Report 506, INRIA, January 1990.
- [CF78] L. M. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, 27(12):1112–1118, December 1978.
- [DSB88] M. Dubois, C. Scheurich, and F. A. Briggs. Synchronisation, coherence and event ordering in multiprocessors. *IEEE Computer Survey, Tutorial Series*, pages 9–21, February 1988.
- [JLHB88] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.
- [KEW<sup>+</sup>85] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, and R. G. Sheldon. Implementing a cache consistency protocol. In *Proc. of 12th Annual International Symposium on Computer Architecture*, pages 276–283, Boston, 1985. IEEE.
- [Lam81] B. Lampson. Atomic transactions. In *Distributed Systems and Architecture and Implementation : an advanced course*, volume 105 of *Lecture Notes in Computer Science*, pages 246–265. Springer Verlag, 1981.
- [Mic90] B. Michel. Gothic memory management : a multiprocessor shared single level store. Technical Report 1202, INRIA, March 1990.
- [Mor90] C. Morin. Building a reliable communication system using high speed stable storage. In *Proc of the 5th Int. Symp. on Computer and Information Sciences*, Cappadocia, Turkey, November 1990.
- [RAA<sup>+</sup>88] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, P. Léonard, S. Langlois, and W. Neuhauser. The Chorus distributed operating system. *Computing Systems*, 1(4), 1988.
- [Roc90] B. Rochat. Implementation of a multi-cache system on a loosely coupled multiprocessor. In *Proc. of 5th Distributed Memory Computing Conference*, pages 676–681, Charleston, April 1990. IEEE.
- [TR85] A. Tanenbaum and R. Van Renesse. Distributed operating systems. *ACM Computing Surveys*, 17(4):419–470, December 1985.

# Supporting an object-oriented distributed system: experience with Unix, Mach and Chorus

*F. Boyer, J. Cayuela, P.Y. Chevalier, A. Freyssinet, D. Hagimont*

Unité Mixte Bull-IMAG/Systèmes, 2, avenue de Vignate, 38610 Gières, France -  
Internet: [guide@imag.fr](mailto:guide@imag.fr) - Phone: +33 7651 7879

## Abstract:

This paper describes our experience in the implementation of the Guide object-oriented distributed system on Unix, Mach and Chorus. A full version of Guide has been developed on Unix. While a full micro-kernel based version of Guide does not yet exist, the basic mechanisms for the support of the Guide virtual object memory and computational model have been implemented on Mach and Chorus. The paper reports the preliminary results of this experience and provides an evaluation of the adequation of micro-kernel technology, as compared to Unix, for the support of object-oriented distributed systems. Microkernels provide a great modularity and flexibility for the conception of an operating system with the server architecture. Execution of lightweight activities, efficient communication and distributed shared virtual memory are very useful functionalities for managing the distribution.

**Keywords:** Object-oriented distributed systems, Mach, Chorus, microkernels

## 1 Introduction

This paper describes our experience in implementing the object-oriented distributed operating system Guide (Grenoble Universities Integrated Distributed Environment) on top of two micro-kernels, Chorus and Mach. Based on this experience, we compare these two implementations with a previous one developed on top of the Unix system [1] [2].

The aim of the Guide project is to explore distributed computing, structured in terms of objects and based on a set of heterogeneous workstations interconnected via a local area network. The system is targeted towards cooperative applications such as document processing and program development. Therefore, *object sharing* is an important feature of our model: objects are persistent and are the principal way of communication between concurrent activities. The project aims to define an architecture which hides most of the problems inherent to distribution, for storage, execution and resource allocation, thus providing location, access and execution transparency. The object-oriented model allows us to integrate several concepts related to operating systems, programming languages and databases, in a single approach.

Guide is a component of Comandos (Construction and Management of Distributed Open Systems), a project supported by the Commission of European Communities under the ESPRIT program.

The first Guide prototype was implemented on top of the Unix system. This implementation aimed to provide rapidly a version of the system capable of supporting simple distributed applications based on the object model. This goal has been achieved and the current version of Guide supports several experimental applications such as document editing, mailing and distributed diary.

While the Unix implementation of Guide provides an adequate testbed to investigate the development of distributed applications, it suffers from several limitations, both in functionality and in performance. More precisely, we identified a lack of adequate support for the following critical aspects:

- shared objects,
- synchronisation mechanisms,
- lightweight activities,
- communications.

In the recent years, a new organization has emerged for operating systems, in which a "micro-kernel" provides the basic functions of memory management, process scheduling and communication, whereas more elaborate services are implemented as specialized servers. Two representatives of this new organization are Chorus [3] [4] and Mach [5] [6]. We expect that implementing the Guide object-oriented architecture on top of one of these kernels would improve over the Unix prototype both in performance and in functionality. In addition, we hope to improve the modularity of the system, and to be able to experiment with different kind of architectures. In this way, a version of Guide has yet been implemented both on top of Mach and Chorus, which is sufficient to evaluate these improvements.

The objective of this paper is to report on this experience. We give a preliminary assessment of the adequacy of the microkernels for the support of an object-oriented distributed system and we compare them, in this respect, with Unix. More precisely, the discussion is focused on the mechanisms that allow to implement a virtual object memory in which shared objects are the main support for communication between concurrent activities or applications.

We are aware of two related recent experiments aimed towards providing an object-oriented distributed environment on top of Chorus and Mach, respectively: the COOL project at Chorus Systèmes [7] and the MachObjects approach at Carnegie Mellon [8]. COOL and MachObjects have a very similar functionality. In both systems, applications are structured in terms of object servers responding to client requests. As opposed to Guide, both systems support an active object model, i.e. objects are the units of sequential execution management. As a consequence, neither COOL nor MachObjects provides a mechanism for object sharing. However, they achieve an equivalent function by allowing memory segments to be shared between objects. No explicit support is provided for persistent objects.

The rest of this paper is organized as follows. Section 2 is an overview of the Guide object-oriented model. Section 3 is a survey and a comparison of Chorus and Mach functionality, with special emphasis on the mechanisms for virtual memory support and activity management. Section 4 describes and compares the implementations of Guide on Unix and on both micro-kernels and presents the lessons learned in this experience. Conclusions and perspectives are presented in section 5.

## 2 Overview of the Guide object-oriented model

This section provides a general description of the architecture of the Guide system, independently of any specific implementation. We first present the object-oriented model which is the basic foundation of the system. We then describe the computational model, which defines the organization and interaction of activities during the execution of an application. We finally present the object storage subsystem, which is in charge of the storage of persistent information.

### 2.1 The object-oriented model

The choice of an object model for Guide is the most important design decision of the system. Objects not only provide a convenient and powerful means for application structuring; they also allow to unify the concepts of procedural and data abstraction, execution units and long-term storage units.

An object encapsulates data (the state of the object) and operations, also called methods. The data may only be accessed through method invocation. A type describes a common behavior that is shared by all the objects of that type. This behavior is defined by the signatures of the methods. A class defines a specific implementation of a type: it contains the internal description of the representation of the data and the programs of the methods. A class is used to generate instances, i.e. objects whose representation and methods are defined by the class.

Objects are not only the units of applications structuring, but also the support for permanent storage. Objects are said to be *persistent*, i.e. their lifetime is not related to that of the execution unit in which they were created. The functions of a traditional file system are subsumed in the persistent object store.

Within the system, objects are named by low-level, location-independent identifiers called *references*. References are used internally for object invocation; they also provide a way to refer to objects within other objects, thus allowing to build complex, possibly distributed, structures.

The object model is accessible to users through a specific language [9] whose run-time system is implemented on top of the Guide kernel. This language allows to use all the concepts of the Guide object model.

### 2.2 Execution management

The computational model provided by Guide aims to allow a user to control the concurrent execution of the activities of his application, while preserving location transparency. The user has the vision of a *virtual machine*, which hides the details of distribution, but which provides mechanisms for concurrency control. The system implements this virtual machine by sharing the load between the nodes of the distributed system.

#### 2.2.1 Parallelism and synchronisation

The main abstraction provided by the computational model is called a *job*. A job is defined as a multi-processor, transparently distributed virtual machine, which groups together in a common address space a set of objects and a set of threads of control, also called *activities*, operating upon these objects. Guide objects are passive, in so far as they are dissociated from threads. The composition of the virtual machine can evolve dynamically. The execution of an activity within a job consists of a sequence of invocations of methods on objects.

To express concurrency, an activity may create at any time a set of child activities which are executed in parallel within the same job, through a COBEGIN-COEND construct. The parent



activity is suspended until a termination condition is satisfied (e.g. termination of the first child or termination of all children, etc).

### 2.2.2 Object sharing

The notion of object sharing is central in the Guide model. Shared objects provide the main communication facility between activities, within a single jobs or between different jobs.

The set of objects within a job at a given time is called the *context* of the job. The job context is implicitly shared between all the activities of the job. In addition, objects may be shared between the contexts of two jobs. For example, in Figure 1, the two jobs *J1* and *J2* share the object *d*.

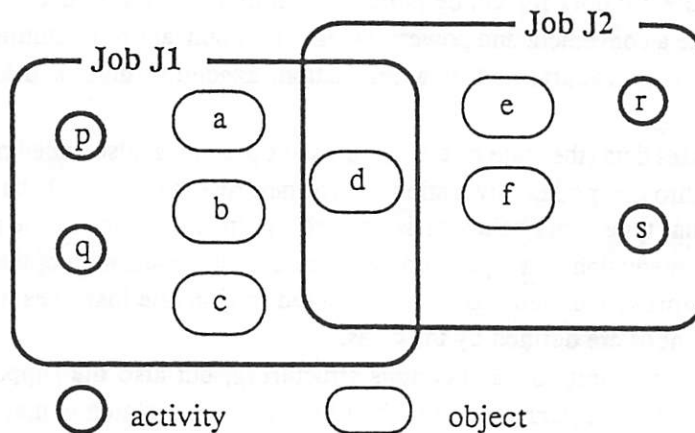


Figure 1. Jobs and activities

Object sharing must be controlled in order to ensure that the shared data remain in a consistent state. This is achieved by synchronization constraints associated to each method of a shared object. Thus, the data of an object may be accessed according to a specific policy such as readers and writers policy, mutual exclusion or a more complex, application dependent, policy. This synchronisation mechanism is not further described in this paper (see [9] for more details).

### 2.2.3 Distribution

The virtual machine defined by a job is potentially distributed, i.e. a job may be represented on a set of nodes. Since execution transparency is a major goal of the project, there is no direct link between jobs and nodes. A job can be represented on several nodes and a node can be visited by several jobs. After its creation, a job is only represented in one node, its creation node. Guide provides a *diffusion* mechanism, which allows a job to dynamically extend to several nodes. Thus a job may be viewed as a set of representatives, also called *local jobs*, one on each node on which the job has diffused. The diffusion mechanism is an important feature for the implementation of the execution model, in so far as it allows to exploit the parallelism offered by the network within a job, and allows a great flexibility for distributed resource management.

Figure 2 illustrates job distribution for two jobs, J1 and J2. J1 is entirely represented on node N1, whereas J2 is composed of two local jobs J2/N1 on node N1 and J2/N2 on node N2.

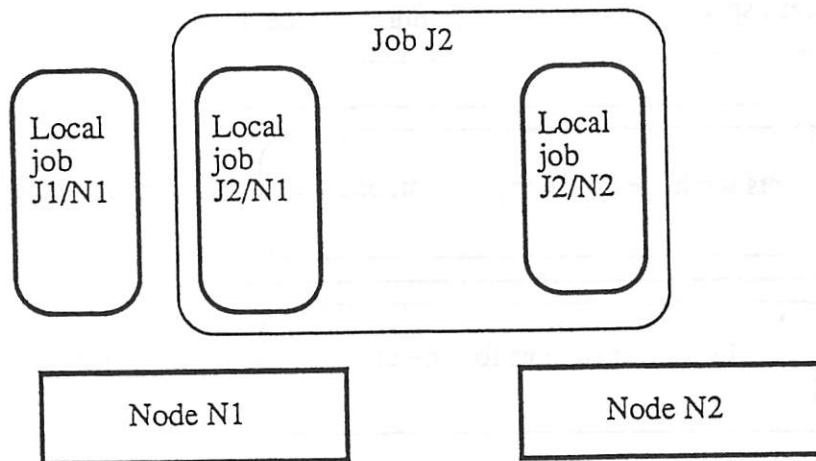


Figure 2. Distribution of jobs

### 2.3 Object management

We make a distinction between two levels of object management: support for object persistence, which is provided by the secondary storage system (SS) and which includes all objects potentially accessible to jobs, and the virtual object memory (VOM) which includes the objects bound to jobs at a specific time. Figure 3 illustrates this architecture.

The secondary storage system is implemented by the cooperation between the set of nodes provided with a secondary memory reserved for permanent storage. The movement of objects between VOM and SS is automatically managed by the system, according to the policy defined for object loading after an object fault. The default policy is as follows: if the invoked object is already loaded on a remote node, execution takes place on that node; if not, the object is loaded on the node on which the object fault occurred.

To conclude this brief survey of the Guide architecture, we summarize its main relevant aspects: applications are organized as a set of transparently distributed "object spaces" in which concurrent activities may be executed; objects may be shared between applications, and between activities within an application; objects are potentially persistent and are mapped from a secondary storage system.

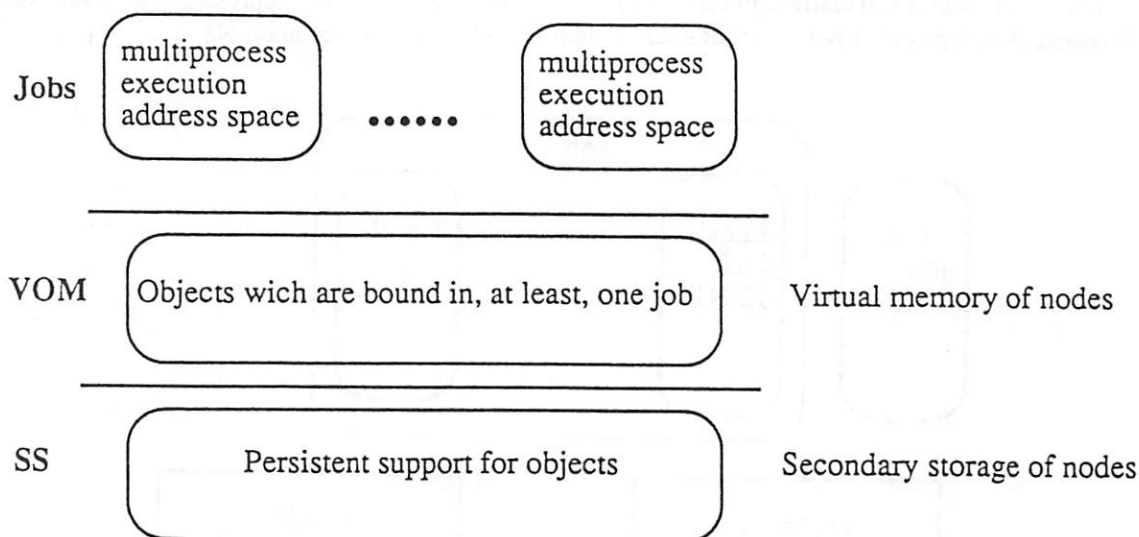


Figure 3. Architecture of the Guide system

### 3 Survey of micro-kernel functionality

The Chorus [4] and Mach [6] kernels supply several similar abstractions which appear to provide adequate support for an object-oriented model like Guide. Before describing our experience with the implementation of Guide on top of Chorus and Mach, we briefly recall the basic concepts of these two systems.

Section 3.1 presents the concepts common to both systems and section 3.2 describes their main differences. Section 3.3 describes the implementation of a shared memory server, a basic mechanism for the support of shared objects. The common concepts and mechanisms are described using the terminology of Mach.

#### 3.1 Common concepts

The first fundamental abstraction is called a *task*: a task is a set of resources such as memory or communication ports. It can be viewed as a virtual address space within which activities are executed.

The second abstraction is called a *thread* and represents a sequential activity. In first approximation, it consists of a processor state and an execution stack. A thread runs within a task; all resources of a task are shared between its threads. Threads are a basic tool for structuring parallel applications. The traditional concept of process is represented by a single thread running within a task.

A *port* is a communication channel.

A *message* is a unit of communication between threads.

Memory management is very similar in both micro-kernels. Memory is managed outside the kernel by a server called a mapper (in Chorus) or an external pager (in Mach). An external pager provides a set of functions that respond to requests sent by the kernel (i.e. page-fault) or by users tasks (i.e.

allocate memory). The main advantage of this memory management architecture results from its flexibility. Thus, a kernel developer can implement his own design in his own external pager as long as he respects the interface between the kernel and the external pagers. An external pager allows the users tasks to get virtual memory within their virtual address spaces. It also provides different mechanisms for memory sharing between tasks such as sharing by inheritance, sharing by mapping, etc.

We were especially interested in using an elaborate external pager, called a network distributed shared-memory server, to implement our object-oriented model. This is developed in section 3.3.

### 3.2 Specific concepts

Although Chorus and Mach have been designed with similar goals, their philosophy is not always identical ; differences appear in the following aspects:

- Naming
- Protection and typed messages
- Functional port addressing and port groups

We discuss these differences and we try to evaluate their impact on the design of a distributed operating system.

The first fundamental difference between Chorus and Mach is the naming scheme. Names in Chorus are network-wide global names called Unique Identifiers (UI). A port, a task or a thread is represented by an UI. Because the naming is global, a task which migrates on another node is still accessible by the same UI. Mach objects (i.e. tasks, threads, message queues) are referenced by local identifiers represented by ports. These local identifiers are meaningless outside of their address space (i.e. their task). The knowledge of ports is controlled by the kernel, in order to provide a strong protection mechanism between tasks. The kernel associates port rights with ports. These rights express the semantics of the access a task has to a port, e.g. send right, receive right and ownership right. Thus, ports may only be exchanged through typed messages, which are controlled and analysed by the kernel. This is a strong constraint for the system designers who do not need protection. On the other hand, Chorus allows to exchange names between entities using files, shared memory and messages, since the knowledge of a UI is assumed to guarantee protection. The kernel is not concerned by these exchanges and any additional protection mechanism is left to the designer of the sub-systems. Thus a system which does not need strong protection between tasks (e.g. a real-time subsystem) does not have to pay for it. This contrasts with the Mach point of view that protection is too important to be left to sub-system designers.

Another difference between Chorus and Mach deals with the manipulation of ports. In Chorus, there is the notion of port group which is not present in Mach. A port group allows the user to regroup a set of ports in a single entity. Threads can send messages to the group by broadcast or functional addressing. Groups are very interesting since distributed services are often supplied by a collection of servers distributed over the network rather than by an unique server. Moreover, a port group provides a mechanism to support fault-tolerance : the failure of a server within a group is transparent to clients who continue to address requests to the group. Mach provides a slightly less flexible mechanism for fault-tolerance: when a server fails, its port receiving rights are transferred to a specified backup server which is now responsible for providing the failed service. The other difference is about the notion of set of ports, wich exist in Mach and not in Chorus. A port set allows a thread to block waiting for a message sent to any of several ports. This mechanism may be very useful for protection and authentification management.

### 3.3 Architecture of a network shared-memory server

A network memory server allows a user to create memory objects, i.e. chunks of memory identified by ports (in Mach) or by capabilities (in Chorus). To address a memory object, a thread maps it within its virtual memory (i.e. the address space of its task). Once an object is mapped, page-faults on this object are treated in the same way as "normal" page faults. The kernel sends a page-fault message to the server to which the object belongs.

A network memory server may be implemented by two main architectures. We define the first one as centralized and the second as cooperative. Both have advantages and shortcomings regarding some specific features.

#### 3.3.1 The centralized architecture

The consistency of an object is managed by an unique server. However, several servers may be active, managing disjoint object sets at the same time. A thread may also access several objects managed by different servers. This architecture is called centralized because all kernels send page-faults on an object to the same server. There is no cooperation between servers to maintain consistency over the network. This architecture is very simple, but it has the main drawback of generating a high network traffic, because all page-fault messages are sent over the network. Figure 4 illustrates this mechanism.

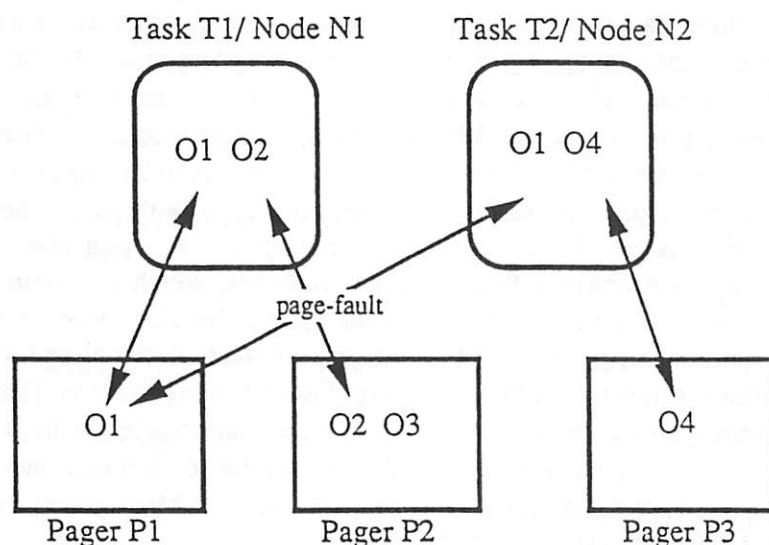


Figure 4. The centralized architecture of a network shared-memory server

Task T1 and T2 share the object O1. The pager P1 is responsible for preserving the consistency of object O1. Each page-fault on O1 is directly transmitted to P1 through the network. There is no cooperation between pagers.



### 3.3.2 The cooperative architecture

Consistency is managed by a set of cooperating servers, one per node. This architecture is more complex than the centralized one, but generates less network traffic because page-fault messages are always local (from the kernel to the server of its node). Each server on each node behave like a cache. Thus only page updates and page invalidations are sent over the network between servers. A complete description of such an implementation is given in [10]. Figure 5 illustrates this mechanism.

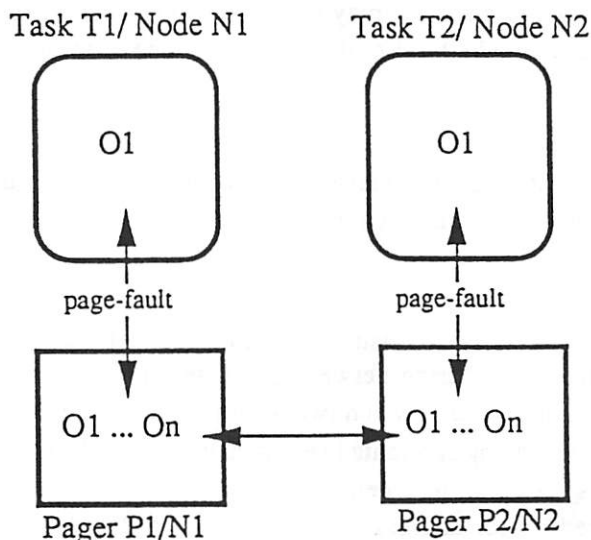


Figure 5. The cooperative architecture of a network shared-memory server

There is one pager per node. Each task access objects through its local pager. Task T1 and T2 share the object O1. Each page-fault is transmitted to the local pager of the task node. Pager P1 and P2 cooperate in order to maintain the consistency of object O1.

## 4 Description of the Unix and micro-kernel Guide implementations

The present section describes the characteristics of the implementations of Guide and compares the implementation on top of Unix with the implementation as a sub-system of a micro-kernel.

The implementation on top of Unix currently runs on the following workstations: Bull DPX 1000 and DPX 2000 running the SPIX system (System V based with BSD extensions), Sun 3-60, Sun 3-80 and Sun 4 running Sun OS (BSD based with System V extensions), and DEC 3100 DecStations running Ultrix.

The implementations on top of micro-kernels is realized on BM 600 running the Chorus V3.2 and Mach 2.5.

### 4.1 Implementing execution structures

There are two major problems for implementing the execution structures of Guide on top of Unix. As described in section 2.2, jobs are distributed, i.e. a job may be represented on a set of nodes. The first major problem is to manage this distributed execution. On the other hand, the activities of a job

share the object space of the job and objects may be shared between different jobs. Therefore, sharing a virtual space between activities belonging or not to the same job is the other major problem.

#### *4.1.1 Implementing distributed structure of execution*

Implementing distributed execution structures is achieved in the same way in Unix and the micro-kernels because neither system directly provides distributed execution structures. Thus a job may be considered as a collection of local components (*local jobs*) on each node on which the job exists. Similarly, an activity, which is a distributed thread of control, can diffuse on another node through a remote object invocation. An activity may be considered as a collection of local components (*local activities*) on each node on which it has diffused ; only one of these local activities can be active at a given time.

#### *4.1.2 Sharing objects between activities*

Sharing objects between activities is not achieved in the same way in Unix and the micro-kernels. The next sections describe and compare the two implementations.

##### *a) Unix implementation*

On top of Unix, there is no basic mechanism to share objects between processes on different nodes. We therefore only allow this sharing between processes on the same node. Thus an object shared between two jobs must be shared between two local jobs on one node.

Several experimental distributed object oriented systems have been built on top of Unix. They have shown three ways for sharing resources between threads on one node. For convenience, we use the term "job" to mean a shared object address space, although a different term is used in each system.

- One process per node. In this solution, all the execution structures on a node are built within one process. This solution is used in Emerald [11]. A scheduler must be provided by the system in the process of each node to manage concurrent threads. All the local activities of all local jobs are in the same address space and data sharing between local activities of one or different local jobs is implicit. This solution has two drawbacks: a scheduler must be supplied, and there is no memory protection between jobs in the same address space. The major advantage is the implicit data sharing.
- One process per local job. In this solution, each local job is implemented by a Unix process. This solution is used in Argus [12]. A scheduler must again be implemented in each Unix process to allow concurrent threads in a local job. Data sharing between local activities in one local job is implicit, because these local activities are threads in the same address space and protection between jobs is guaranteed by the separation of their address spaces. However, if resources must be shared between activities in different jobs, the processes which implement these jobs must share memory. This may be done in Unix by means of the shared memory facility, but protection between jobs may be lost if they share a memory region.
- One process per local activity. In this solution, each local activity is implemented by a Unix process. There is no scheduler to write, but data sharing between local activities of one or different local jobs must again be implemented with shared memory, with the same drawback as in the last solution.

In these solutions, one found two basic mechanisms to share data between threads : implementing a scheduler in a Unix process and using Unix shared memory. The motivation for the choice of the third solution in Guide was to minimize the implementation work for rapid prototyping.

### *b) Micro-kernel implementation*

Micro-kernels provide concurrent threads in the same address space (task). Thus data sharing between threads in one task is implicit. Micro-kernels allow tasks to share data through the mapping mechanism, which provides a cheap way of sharing between threads in different tasks. Then a *local job* is represented by a *task*, while a *local activity* is represented by a *thread* within the task that implements the job to which the activity belongs.

This implementation on a micro-kernel combines all the advantages of the three solutions on top of Unix : a direct data sharing mechanism and an implicit memory protection between jobs. The aspects related to object sharing in virtual memory are further developed in the next section.

## 4.2 Implementing an object virtual memory

### 4.2.1 Implementation on top of Unix

In the Unix implementation of Guide, shared objects are implemented in shared virtual memory. Two solutions may be envisaged for the mapping of shared objects to virtual memory regions.

In the first solution, an object is associated to one shared memory region. This allows to protect object accesses, since an object can not dynamically access another object's memory. This mechanism was used in an early Guide prototype. However, some disadvantages have appeared. The cost of the *attach* and *detach* operations on shared memory data is high. In addition the size of Guide objects is variable since the size of shared memory regions is fixed ; finally the minimum size of a shared region is usually much larger than the average object size.

Therefore, we implemented a second version in which the virtual memory of a node is represented by one single large shared memory region, within which objects bound on this node are mapped. The main drawback of this solution is that it loses the property of dynamic protection between objects. On the other hand, a significant gain in performance was immediately noticed for the binding of an object within the virtual address space (i.e. only one *attach* operation is necessary, when the activity is created on the node). Figure 6 illustrates this implementation.

### 4.2.2 Implementation on top of a micro-kernel

A *job* is now represented by a *task* (cf section 3.1). A task defines a protected address space which is a set of regions. Regions are access windows upon memory objects (also called segments). A memory object may be used as the unit of persistent representation in secondary memory. Thus a Guide object is represented by a region in virtual memory and by a segment in permanent storage. Memory objects are managed by external pagers; they are the units of Virtual Object Memory and Secondary Storage control.

Mechanisms for memory management provided by the micro-kernels appear to be well suited for Guide objects management. They have the following advantages:

- Encapsulating an object within a region provides an implicit protection mechanism for access to objects. Accessing an object does not give access to objects that are in a contiguous address space.
- Threads which represent the activities of a job automatically share the set of objects bound in their job, since the address space of a task is shared by the threads of this task.
- To share an object, activities in different jobs on the same node use the mapping mechanism (i.e. a memory object may be mapped in a set of regions within the same node) provided by an external mapper.

The representation of objects in virtual memory has been presented. Thus the management of object invocation, which introduces the problem of sharing objects between different nodes, is discussed in the next section.

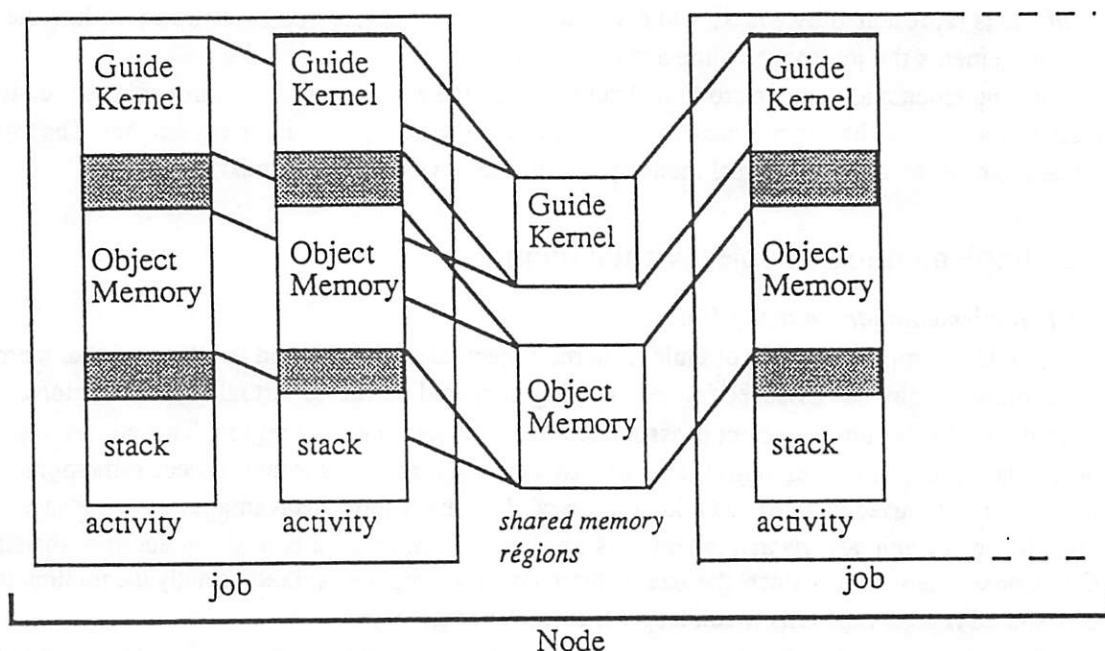


Figure 6. Structure of the object virtual memory on one node with two jobs

### 4.3 Mechanisms for object invocation

#### 4.3.1 Implementation on top of Unix

Two mechanisms may be used for remote object invocation. In the first solution, objects are replicated. Invoking an object on a node always involves a local copy of the object. A replication protocol is needed to ensure that the copies of an object remain consistent. This solution is used in Orca [13]. In the second solution, a single copy is maintained, and invocation always takes place on that copy, either by moving the object to the invoking node or by creating a representative of the calling activity on the node that holds the object and performing the invocation there ("diffusion").

We have used the second solution in Guide, for the two following reasons: first, the small granularity of the Guide objects would make a consistency protocol expensive. Second, the mechanisms of activity diffusion provide additional flexibility which should allow to implement load balancing policies. We now describe the invocation mechanism, as implemented on Unix.

When a method is invoked on a remote object, two cases may occur. If the object is not bound to a job on the remote node then the object is brought on the current node and invocation is performed locally. Otherwise, the *extension mechanism* is used, i.e. the calling activity diffuses to the remote node. A representative of the activity is created on the remote node and the invocation is locally performed on that node. This synchronous invocation scheme is illustrated on figure 7.

The creation of a remote local activity is called *activity extension*. This activity extension can also cause the creation of a local job on the remote node if the job does not exist yet (e.g. as a result of a previous call). This creation of a local job on the remote node is called *job extension*.

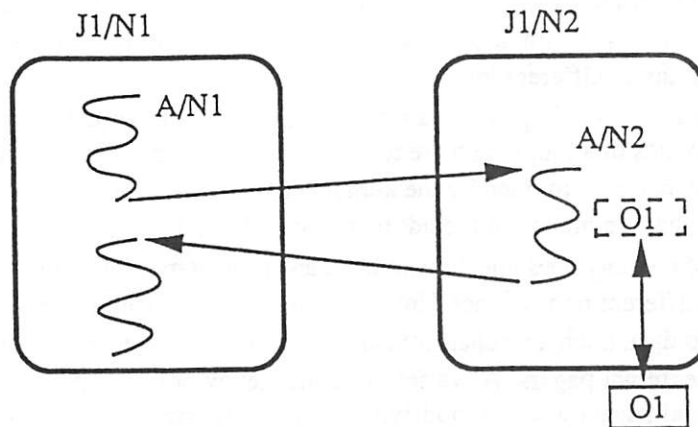


Figure 7. A remote execution with job and activity extension

The process A/N1 which implements the invoking activity is blocked on the initial node N1 waiting for the results of invocation. Then the process A/N2 on the remote node N2 resumes its execution, terminates the method and transfers the results back to the invoking activity.

#### 4.3.2 Implementation on top of a micro-kernel

The mechanisms used for object invocation depend on the strategy used to manage memory objects. The first strategy consists to use an external pager which constrains a segment to be located on a single node. In this case, a remote invocation is done using the same extension mechanism as in the Unix implementation (the object migration can also be used). The external pager which implements the second strategy allows an objects to have copies on several nodes, thus implementing a Distributed Virtual Object Memory (DVOM). In this case, all invocations are local. This simplifies object management, but the cost of maintaining consistency must be considered. This cost depends on the behavior of the applications that the system supports, and especially on how they use shared segments (i.e read to write ratio).

The extension mechanism and object migration may be used concurrently with the DVOM, e.g. to use the potential for parallelism provided by multiple work-stations or to implement load-balancing over the network. The choice of the strategy can be left to the user or controlled by the system..

## 4.4 Lessons learned

In this section, we summarize our comparative experience in implementing the basic object management mechanisms of Guide on Unix and on the Mach and Chorus micro-kernels. We consider the following aspects:

- Distributed object memory
- Persistent memory
- Global naming
- Fault-tolerance



### *Distributed object memory*

As mentioned in section 4.2, the Guide implementation on Unix uses shared memory to allow multiple flows of control (i.e. activities) to run within the same address space. The Virtual Object Memory on a node is implemented using a single shared memory region containing the objects bound on the node. This solution has an important drawback: it does not allow any memory protection between activities belonging to different jobs.

Using Mach or Chorus allows to implement a job by a task. Thus, the virtual memory of the task is shared between the activities of a job, which are represented by threads. This architecture also uses the strong protection mechanism provided by the kernel between tasks, i.e. threads within a task can only access the objects that are mapped in the address space of this task.

Using the Unix shared memory does not allow to build an efficient mechanism for managing multiple copies of an object on different nodes, since Unix does not directly provide any tools to preserve consistency of replicated data. Such a mechanism can be efficiently implemented using micro-kernel functionalities such as external pagers. A variety of consistency policies may be tested using the modularity of the external pagers without modifying the architecture of the operating system. One can choose to use the standard tools provided by the system, or one can choose to develop specific servers for a given application.

In the Guide implementation on the micro-kernels, this architecture is embodied in the concept of a distributed set of shared objects, which we call the Distributed Virtual Object Memory. Using the DVOM, one can exploit real parallelism between activities accessing the same set of objects, since they can execute on different nodes while accessing the objects.

### *Persistent memory*

The management of the page-faults using an external pager can be integrated with the management of the secondary storage for persistent objects. An external pager manages a swap area on the disk to store the pages. This swap area can be used as a storage area for persistent objects. By directly swapping on the objects area, storage management is implicit. Thus, one can expect good performances with such a pager. Using a set of cooperating external pagers, one can also implement a protocol to manage a reliable duplication of the objects storage.

### *Global naming*

In the Unix implementation, a global naming scheme has been explicitly developed using unique global identifiers (uid's). Such a mechanism is directly provided through capabilities in Chorus and ports in Mach. However, the global naming mechanism built on top of Unix also allows object migration through forwarding pointers, which is not provided by the micro-kernels' global naming.

### *Communication*

The management of communication is an important difference between Unix and the micro-kernels. Communication primitives are an essential component of the micro-kernels and they are closely integrated with virtual memory, while they have been added to Unix as an upper layer. In the Guide implementation on top of Unix, we use standard mechanisms such as sockets and messages queues for internal kernel communications, but we have developed our own communication protocol to support a reliable and efficient remote object invocation mechanism. The micro-kernels directly provide such communication functionalities.

### *Fault-tolerance*

The architecture of systems built on top of a micro-kernel has a good potential for fault-tolerance. If a failure occurs, it is local to the server on which it happens. The whole system is not corrupted. Thus, a server replication mechanism or a regenerative algorithm can allow the system to restart. A set of tools for dynamic reconfiguration of the system using functionalities such as port group, port migration or port backup can also be developed. This will be an area for further research.

### *Performance aspects*

The implementation of Guide on top of Mach and Chorus is only at a preliminary stage ; but we can now [january 1990] run distributed applications on Guide on top of Mach 2.5 :

- we didn't modify the Guide compiler,
- we chose a centralized architecture for the shared memory server,
- we provided object migration and activity diffusion mechanisms.

We have 2 years' experience with the Unix implementation and only a few month with the microkernels one. Therefore, we cannot supply significant performance figures at the time of writing. However, we can expect better performances on the micro-kernels because:

- Activities that were implemented with processes in the Unix prototype are now implemented with threads which are light-weight processes.
- Most of distributed protocols which were developed at an upper level on top of Unix are now essential components of micro-kernels (communication, naming).

## 5 Conclusion

We have presented our experience in implementing the object-oriented distributed operating system Guide on top of two micro-kernels, Chorus and Mach. The objective of the paper was to assess this experience and to compare it with the first implementation of Guide which has been done on top of Unix for fast prototyping.

The Guide system provides an execution environment for an object-oriented programming language. The main features of the system are: persistent shared objects, supported by a distributed two-level storage, transparent distribution of objects, execution model based on concurrent, distributed jobs and activities, support for synchronisation and transactions.

The first remark about the implementation of a distributed object-oriented system relates to the architecture of the support system. Kernels like Mach and Chorus are organized as a set of servers which are managed by a minimal micro-kernel. Such an organization provides a large facility and flexibility for implementing distributed systems. The modular aspect allows to experiment separately different parts of the system, and to evaluate different strategies.

Secondly, micro-kernels integrate basic mechanisms for distribution. Most services which had to be built at an upper level in the distributed systems of the Unix generation, are directly integrated in the micro-kernels. For example, Mach and Chorus provide a location independent global naming scheme. Thus, object naming is more efficient and easier than on Unix. Mechanisms for remote procedure call are also directly provided by micro-kernels. Thus, Guide remote object invocation can be readily supported.

We finally notice that some functionalities provided by micro-kernels allow to implement strategies that cannot be implemented on top of Unix. For example, micro-kernels directly support distributed virtual memory management. Coherence preserving is ensured by the kernel. For the Guide system,

this method allows to implement a Distributed Virtual Object Memory that seems to be more efficient than the object management strategy on Unix, which relies on remote object invocation with a single object image. In the same way, micro-kernels allow to implement easily fault-tolerance protocols with the possible data replication or by using the notions of port and port group, which dissociate server location and server naming, to implement reconfiguration algorithms.

On the other hand, while it appears to be easier to implement distributed systems on top of micro-kernels than on top of Unix (since most of the mechanisms that are required for distribution management are directly provided), it seems to be more difficult to exploit all the power of these mechanisms. For example, the notion of a port group should allow us to represent a job as a distributed entity and to use group functionality to implement job mechanisms. Execution structures that efficiently use this mechanism are still to be explored. Another example concerns object management. As we noticed, micro-kernels allow to share objects on different nodes using the DVOM. Thus, all object invocations can be made on the local image of the object. However, Guide also provides the job extension mechanism which allows a job to perform efficiently a remote object invocation while keeping a single object image. Further study, including performance evaluation, is needed to compare these two mechanisms and to design strategies that use the most appropriate mechanism according to the conditions.

### Acknowledgments

We would like to thank Professor Sacha Krakowiak for his help in reviewing this paper, and Frederic Dajean for helping us in the implementation of DVOM on top of Mach.

This work was done with the support of the Open Software Foundation Research Institute in Grenoble ; special thanks to Philippe Bernadat for his assistance.

The Guide project is supported by the Commission of European Communities through the ESPRIT program in project COMANDOS (Construction and Management of Distributed Open System), the Universities of Grenoble (Institut National Polytechnique de Grenoble - Université Joseph Fourier) and Centre National de la Recherche Scientifique.

## Bibliography

- [1] D. Decouchant, A. Duda, A. Freyssinet, M. Riveill, X. Rousset de Pina, R. Scioville and G. Vandôme. Guide: an implementation of the Comandos object-oriented architecture on Unix. *Proc. EUUG Autumn Conf.*, (Lisbon), pp. 181-193, oct. 1988.
- [2] D. Decouchant, M. Riveill, C. Horn and E. Finn. Experience with implementing and using an object-oriented, distributed system. *Proc. Usenix Workshop on Experiences with Distributed and Multiprocessor Systems.*, (Fort Lauderdale), pp. 301-310, oct. 1989.
- [3] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard and W. Neuhauser. The Chorus distributed operating system. *Computing Systems*, 1(4), pp. 305-370, dec. 1988.
- [4] *Chorus Distributed Operating System*. Chorus Systèmes, feb. 1989.
- [5] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian and M. Young. Mach: A new kernel foundation for Unix development. *Proc. Summer Usenix Conference*, pp. 93-112, july 1986.
- [6] R. Baron, D. Black, W. Bolosky, J. Chew, R. Draves, D. Golub, R. Rachid, A. Tevanian and M. Young. *Mach Kernel Interface Manual*. School of Computer Science, Carnegie Mellon University, sep. 1988.
- [7] S. Habert, L. Mosseri, V. Abrossimov. COOL: Kernel support for object-oriented environments *Proc. OOPSLA'90*, (Ottawa), oct. 1990.
- [8] D. Julin. *Mach Objects Reference Manual*. School of Computer Science, Carnegie Mellon University, aug. 1989.
- [9] S. Krakowiak, M. Meysembourg, H. Nguyen Van, M. Riveill, C. Roisin and X. Rousset de Pina. Design and implementation of an object-oriented, strongly typed language for distributed applications. *Journal of Object-Oriented Programming.*, 3(3), pp. 11-22, sept.-oct. 1990
- [10] A. Forin, J. Barrera, M. Young and R. Rashid. *Design, Implementation, and Performance Evaluation of a Distributed Shared Memory Server for Mach*. School of Computer Science, Carnegie Mellon University, aug. 1988
- [11] A.P. Black, N. Hutchinson, E. Jul and H. Levy. Object structure in the Emerald system. *Proc. OOPSLA'86*, Portland, Oregon, ACM SIGPLAN Notices, 21 (11), pp. 76-86, nov. 1986.
- [12] B. Liskov, D. Curtis, P. Johnson and R. Scheifler. The implementation of Argus. *Proc. 11th Symp. on Operating System Principles*, ACM SIGOPS Review 21(5), pp. 111, 122 November 87.
- [13] H. Bal, M.F. Kaashoek and A.S. Tanenbaum. A distributed implementation of the shared data model *Proc. Usenix Workshop on Experiences with Distributed and Multiprocessor Systems*, (Fort Lauderdale), pp. 1-20, oct. 1989.

The first part of the paper discusses the background and motivation for the research. It highlights the challenges associated with distributed systems and the need for a more efficient and scalable solution. The authors then present their proposed approach, which is based on a novel algorithm for resource allocation and scheduling. This approach is designed to optimize system performance while minimizing resource usage and ensuring fairness among users. The paper includes a detailed description of the algorithm, its implementation, and the results of experiments conducted to evaluate its effectiveness. The authors conclude by discussing the implications of their findings and the potential for future work in this area.



# Can We Study Design Issues of Distributed Operating Systems in a Generalized Way?

---

## RHODOS\*

G.W. Gerrity, A. Goscinski, J. Indulska, W. Toomey, W. Zhu

Department of Computer Science  
University College  
The University of New South Wales  
Australian Defence Force Academy  
Canberra, ACT

### Abstract

RHODOS is a distributed operating system under development at the University College, University of New South Wales, for generalized research into the problems of distributed systems. It provides a transparent, distributed system with attributed system names, fast and flexible interprocess communication, and resource migration for system performance. RHODOS allows the study of different methods and algorithms for naming, load balancing and process migration, communication subsystem supporting process migration, portable memory management necessary for heterogeneous environments, protection, communication security and authentication, and their influence on the overall system performance.

\* This work was partly supported by the Australian Research Council under Grant A48831034, and the Australian Research Council Fellowship Scheme under Grant 890270.

George Gerrity, e-mail:  
Andrzej Goscinski, e-mail:  
Jadwiga Indulska, e-mail:  
Warren Toomey, e-mail:  
Weiping Zhu, e-mail:

gwg@csadfa.cs.adfa.oz.au@uunet.uu.net  
ang@csadfa.cs.adfa.oz.au@uunet.uu.net  
jaga@csadfa.cs.adfa.oz.au@uunet.uu.net  
wkt@csadfa.cs.adfa.oz.au@uunet.uu.net  
wpz@csadfa.cs.adfa.oz.au@uunet.uu.net

# 1 An Introduction to RHODOS

Existing distributed operating systems at the moment seem to be experimental; none of the available systems are of commercial quality (Mach is probably the most advanced contender, but it still fights for recognition of its quality). The existing systems were developed on the basis of quite different approaches, and this implies that they are not easily comparable. The developers of these systems have given little explanation as to how they arrived at their design decisions, and consequently, there has not been wide discussion on the merits of alternative design philosophies.

Users of distributed systems can complete their computational tasks faster with lower processing costs because distributed operating systems allow them to share resources, provide them with a great variety of services, and allow use of sophisticated devices very often not compatible with their computers and not accessible within their organizations.

The first set of problems can be solved by building a test-bed system, which will allow comparisons between different models and implementation techniques in the area of different components of distributed operating systems. The second aspect can be solved by the creation of a kernel which can be used to support both distributed operating systems and network operating systems. Our goal is to attack these two problems uniformly.

There are two basic goals of the Research Oriented Distributed Operating System called RHODOS. The first is to allow investigating and comparing alternative contending structures and methodologies for implementing the components of a distributed operating system. This study will initially be carried out for homogeneous distributed systems. The results of these investigations will then be integrated to construct a prototype homogeneous distributed operating system. This will be repeated for a heterogeneous distributed operating system. This leads to the second goal which is to be a platform for the development of operating systems for Open Distributed Processing.

To achieve the first goal of RHODOS, i.e., building a system which allows both qualitative analysis and quantitative comparisons, it must sit on top of the bare machine. This allows to see all resources without interference from any part of an old centralized operating system.

The aim of this report is the presentation of the basic logical and detailed design features of the homogeneous RHODOS system. Section 1.1 and 1.2 gives a brief look at the design issues of the system. The basic internal structure of RHODOS is discussed in Section 2. RHODOS' objects and their naming are presented in Section 3. In particular, system names are discussed. Process types are dealt with in Section 4. Interprocess communication, vital in a distributed system, is presented in Section 5. Section 6 discusses the portable approach to memory management that has been used in RHODOS. Section 7 contains the detailed design of load balancing in RHODOS. In particular, implementation aspects of the load balancing facility, the interface between the load balancing server and migration manager, and the migration manager are discussed. Section 8 addresses general aspects of the RHODOS file server. The last section concludes with comments on future work.

## 1.1 Design Issues

Based on experience in the research and development of distributed operating systems, it is suggested here that RHODOS should be developed on the basis of a distributed kernel.

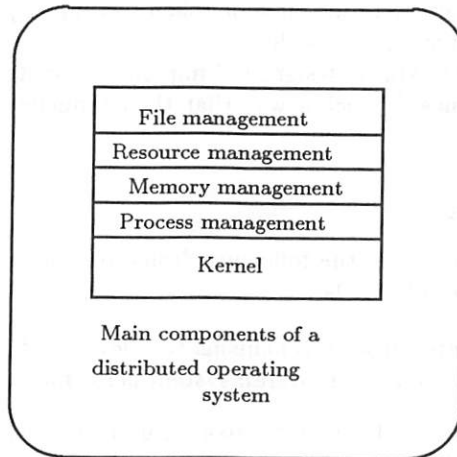
Moreover, RHODOS contains the same management components as an operating system for centralized systems, i.e., process management, memory management, resource management, and file management. The development of these components requires study of many issues to find methods and algorithms which can be used to build these components.

The problem is how to develop such a distributed kernel. This problem must be solved in a wider context. We suggest here that the development of RHODOS requires taking into consideration of several design issues.

Two groups of design issues have been identified [Goscinski 1991]. To the first belong issues generated by the requirement of users and their applications, *openness* and performance. These issues are: a communication model for an operating system, paradigms for process interaction in a distributed computing environment, transparency, heterogeneity, autonomy and/or interdependence, reliable distributed computing, replication of information and data consistency.

### User and Performance Oriented Issues

Communications model  
Paradigms for  
process interaction  
Transparency  
Heterogeneity  
Autonomy and/or  
interdependence  
Reliable computing  
Replication of information  
and data consistency



### Components-oriented Issues

Interprocess communication  
Synchronisation  
Addressing and naming  
Process management  
Resource allocation  
Deadlock detection  
and resolution  
Communications security  
and authentication

Figure 1: Design Issues

In the second group are those which relate directly to the components of a distributed operating system; this comprise issues oriented towards distributed processes and their management including: interprocess communication, synchronization, naming, and process management; and also issues oriented towards resource management and addresses such as: resource allocation, deadlocks, protection, communication security and authentication, and services. These two groups of issues are illustrated in Figure 1.

The first issues is a communication model for a distributed operating system. The ISO/OSI Reference Model can only be treated as a basic framework for message passing systems. It requires passing information down and up through all layers on the sending and receiving computers, respectively, and it is too complicated to be applicable for distributed operating systems. A functional hierarchy based model should be used.

Paradigms for process interaction in a distributed computing environment form the second issue. The following basic paradigms are identified: the client-server model, the integrated model, the pipe model, and the remote procedure call model. Performance and reliability requirements lead toward the client-server model and the remote procedure call model.

A distributed operating system ensures that users view a distributed system as a virtual uniprocessor, not as a collection of distinct machines connected by a communication system. Moreover, users do not see any difference between local and remote resources i.e. geographic distribution is invisible. This is possible due to network transparency. Various degrees of network transparency can be required to achieve a distributed operating system: access, location, name, control, data, execution, and performance. Transparency is closely associated with homogeneity.

However, both some applications and external circumstances (e.g., administrative, managerial) generate the need for heterogeneity. This issue generates process, name and resource management problems. Heterogeneity will be dealt with when studying open operating systems.

Transparency and heterogeneity are closely associated with the next issue, autonomy and interdependency, which are achievable. Users want a distributed operating system to provide autonomy to achieve policy freedom, robustness, and security. On the other hand, interdependence allows autonomous computer systems to communicate, decreases overall costs of distributed systems, makes possible efficient utilization of resources, and improves performance. This implies the need for a trade-off between autonomy and interdependence. However, the problem is what is such a trade-off.

Performance and reliability in distributed systems can be greatly improved by providing redundant resources on different nodes. However, resource redundancy generate some serious problems, such as the lack of the global state information, the possibility of partial failures, etc. This implies difficulties in maintaining data consistency. Multiple copy update can be carried out based on two

solutions: non-voting, using either prime sites or token passing approaches, or voting, using either majority voting or weighted voting approaches.

This project is an attempt to study, design, and implement a distributed operating systems taking into account these design issues, in such a way that their influence on overall system performance can be studied.

## 1.2 Design Decisions

In the light of the discussion above, the following design decisions have been made for RHODOS [Gerrity et al. 1988, Gerrity et al. 1990a]:

- Policy should be separated from mechanisms to allow an efficient study of variety of policy decisions and some mechanisms on overall system performance.
- There is a freely available pool of services to each user process, such as a file service etc. These are provided by a number of *server* processes which take requests from user processes and return the desired result. The user processes hence are clients to the services.
- RHODOS will provide a multitasking and multiuser environment. Several users will be able to use each machine in the system, and the number of active processes may exceed the number of CPUs. Initially the system will be composed of single-CPU homogeneous machines, but RHODOS will be designed with a heterogeneous system in mind, including multiprocessor machines.
- Load balancing and migration are to form an important part of the system. Ensuring that the machine load is similar on all machines, by migration of processes and objects, will improve the system performance and efficiency.
- Resource management and protection also improve the use of the system. At the moment, it is not known whether a centralised or distributed approach will be taken.
- The communications subsystem of RHODOS will be based on RRDP, an efficient transport protocol developed for RHODOS, on top of IP over Ethernet, for communication in an RHODOS environment, and with other systems supported by the DoD protocols (TCP and/or UDP).
- Objects in the distributed system will have attributed names, for protection and identification purposes. Moreover, there will be a hierarchy of names: User names, System names and Locations (Physical Names).
- The majority of RHODOS objects will be totally distributable. They can be accessed from any node in the system, and may well migrate from node to node to improve system use. Therefore the location of each object must be made transparent to the user.

## 2 RHODOS' Internal Structure

RHODOS has been designed to be a very modular operating system, as this helps to easily alter the configuration of the system, and to measure the performance differences between different system policies and different mechanisms. This modularity is reflected by the fact that RHODOS is composed of a co-operating set of independent processes, rather than one monolithic kernel, as shown in Fig.2.

RHODOS is broken into four logical layers of service. The **user** level contains processes that are run by users to do their work. The **server** level has server processes that provide access to the system objects in RHODOS (files, names, resources etc.). Management of the operating system's resources (processes, memory, ports) is done in the **kernel** level. Finally, the mechanisms for interrupt handling, local interprocess communication, and crude process context switching are done at the **nucleus** level.

Because of the distributed nature of RHODOS, a copy of the kernel and nucleus resides on each computer. The presence of server processes on several computers depends on the devices attached to these computers, and the management of the RHODOS system.

### 2.1 User Processes

RHODOS only provides an environment where useful work can be done. This work is performed by **user processes**, which use the services of the operating system. They are protected against each other by RHODOS, and obtain services (e.g. file manipulation or interprocess communication) through the **library calls**, which map requests into messages implemented by **system calls**, a set of routines which are linked in to the user processes at compile time. Library calls are differentiated from system calls in that the latter always cause a software trap to occur, which is handled by the nucleus.

### 2.2 Servers

A lot of the services in RHODOS can be performed without direct access to the data structures kept in the RHODOS kernel. These are handled by **system servers**, which take interprocess requests from other processes (called **clients**), and return one or more messages with the result. Servers are able to deal with requests from other machines, so that a network distribution of system services is possible.

The main servers are:

1. **Name Server.** This server maintains information about the well-known objects throughout the distributed system, and maps the User-name (or UName) of RHODOS objects onto their System-name (or SName) [Vance and Goscinski 1989]. This server should provide not only the conventional object name - location mapping but also white page and yellow page services.
2. **File Server.** The File Server provides file operations on local or remote files, and enforcing appropriate protection on the files. Initially, the NFS file server will be used. An original file server, supporting transactions, distribution, and multiple copy update, will be developed in the second stage of this project.
3. **Load Balancing Server.** As RHODOS is a distributed system the migration of processes from heavily loaded machines to lightly loaded machines is possible. This server collects statistics about the local load from the Nucleus, and makes decisions about the *when* and *where* to migrate *which* processes. If a process is chosen to be migrated, the server negotiates the migration with a remote Load Balancing Server (if a Load Balancing algorithm uses negotiation), and finally requests the Migration Manager to move the process to the remote machine.
4. **Authentication Server.** The Authentication Server allows both:
  - the receiver to determine that the message was sent by the genuine sender, and that the contents of the message were not changed when it was transferred, and



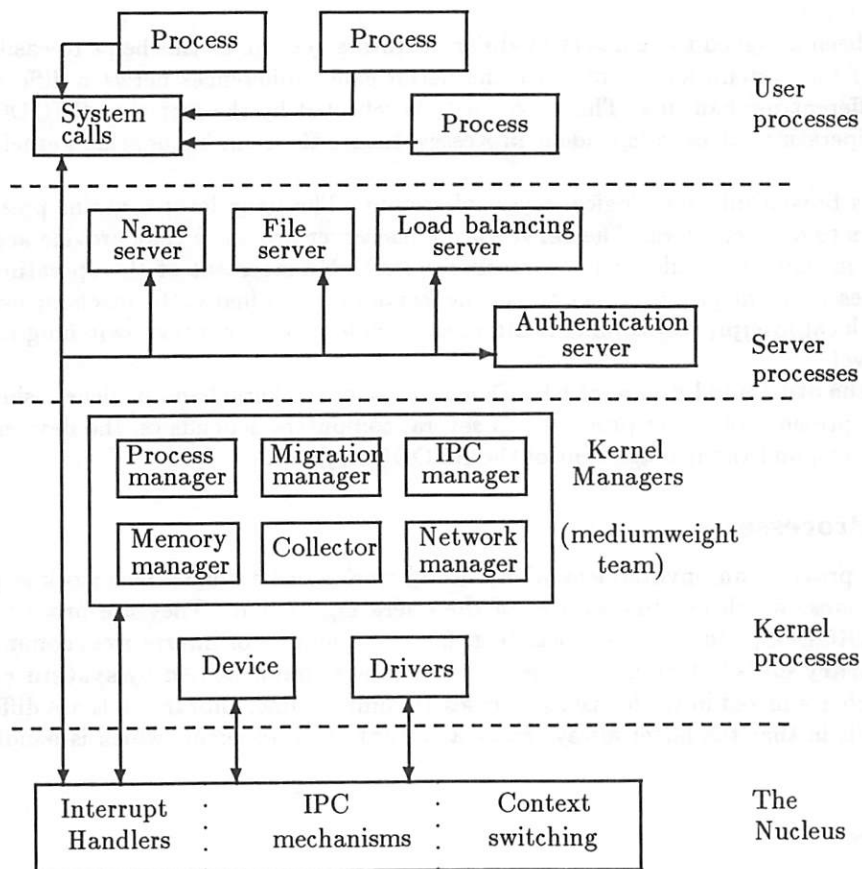


Figure 2: The Process Layers of RHODOS

- a sender to determine whether the message sent was received by the intended receiver, and if the contents of the message were changed.

## 2.3 The Kernel

In monolithic operating systems, all of the services in the system are kept as a set of subroutines in one large file, which is called whenever a system interrupt occurs. This is unsuitable for research purposes, as the system must be recompiled if a change is made, and no changes can be made to the system as it is running.

In RHODOS, apart from primitive interrupt handling, primitive IPC, and context switching, everything else is handled by separate processes. Many of these processes need to share data structures, so a large proportion of RHODOS is formed into a team of mediumweight processes known as the **Kernel**, or the **Kernel Servers**. Such things as process management, memory management, remote interprocess communication, migration management, data collection and fast and reliable delivery are dealt with by the Kernel.

Software interrupts caused by system calls are mapped into local messages, and passed on to one of the Kernel servers for action. It returns the result to the process that caused the interrupt.

The policies of management in RHODOS are implemented in the kernel servers, and the implementation routines in the Nucleus. This means that changing the configuration of RHODOS can

be done by starting servers with different policies at boot time. This allows a flexible comparison of different operating system strategies, which is needed in a research-based system.

The main kernel servers, or managers, at the moment are:

1. **Process Manager** The responsibility of creating new processes, changing their priority, and local process scheduling is done by this process.
2. **Memory Manager** Allocation and deallocation is done in this process, as well as virtual memory operations and paging. The manager is subdivided into three sections which deal with address spaces, segments and pages of memory.
3. **IPC Manager** Interprocess communication which requires a high quality of service, or is intended for a remote machine, is passed to this process, which implements communication primitives, and deals with address resolution and transparency of migration for communication purposes.
4. **Migration Manager** This server deals with process migration requests from the Load Balancing Server. By interacting with its equivalent on another machine, processes are migrated and restarted on the other machine.
5. **Data Collection Manager** This server collects information on local computation load and local and remote communication patterns for each process. This information is passed periodically or when a special event happens (e.g., a process is killed, a new process is created) to the Load Balancing Server.
6. **Network (Message Delivery) Manager** This server is responsible for reliable delivery of messages to and from remote processes. It implements the RRDP protocol, supported by IP.

The Kernel has higher privileges than user processes, as it is able to do such things as kill processes, allocate memory etc. Most of the device drivers in RHODOS also run with high privileges, and although not part of the Kernel Servers, are grouped with them.

## 2.4 The Nucleus

The Nucleus in RHODOS is not a process in its own right. It is executed only when a system interrupt occurs; it deals with the interrupt by manipulating its data structures and passing execution to an appropriate process. For example, system calls are converted into local messages, which are placed on the port queues which exist in the nucleus.

The mechanisms for RHODOS' policies are placed in the Nucleus, so that they are not duplicated, and that access to the mechanisms can be protected. Local interprocess communication is also handled by the nucleus, providing fast and efficient implementation of system calls, and client/server requests.

## 3 Naming and Protection

In every operating system it is convenient for the user to have a name for an object (file, port, etc.), but the operating system needs to have its own naming method with names of fixed length and of a known form. This implies the problem of selecting a user naming method, a system naming method, and the mapping between them.

In a distributed operating system it is also desirable that names be location-independent, but at a very low level, when inter-computer communication is involved, this location-independent name must be mapped to a physical location. Thus, in RHODOS, there is a three-level structure of names and a mapping between levels:

- User Name (textual)
- System Name (fixed length)

- Low Level Name (communication)

The system name (SName) of objects is very important in an operating system and in the case of a distributed system may have an influence on interprocess communication and process migration performance, as well as on the protection system. RHODOS, being designed as a testbed for comparing different solutions in the field of the distributed operating system, should have more than one kind of naming method at the system level.

RHODOS assumes examining of different protection systems: Capability system (sparse capabilities), Access Control List system and a mixture of them. The system name of a RHODOS' object is constructed in the way allowing to build either Capability based system or Access Control List system. A system name of an object is a fixed-length numerical name used by an operating system in order to refer to the object. Every object in an operating system has a number of attributes which describe this object (e.g. the type of the object, the object number, the origin of the object, the creator, etc.). It may be appropriate to include some attributes within the object SName. Such an assumption has been made for the RHODOS operating system [Vance & Goscinski 1989] and the SName of an object includes the type of the object, the origin of the object, the object number and a copy number. In the case of a Capability system, the SName additionally includes access rights to the object. This SName has been extended in order to meet authentication requirements. The extended SName contains some additional fields which can stop its forgery and allow the identification of the owner or the user of an object. This is especially important in the case of sparse capabilities, which have many advantages but they suffer from some drawbacks. The most distinctive is that another user can steal it. To protect against this attack an additional field is added (the 'context' field); this field allows the server that manages the referenced resource to check whether a requesting process works on behalf of a legitimate user or the capability was stolen (or passed between users in an uncontrolled way).

At the user name level an attributed naming system has been proposed. Thus, RHODOS provides not only mapping from a user name to the system name but also white page and yellow page services.

## 4 Processes

Processes in RHODOS are composed of three spaces in virtual memory: the *text* space which contains the code of the process, the *data* space which holds its global data, and the *stack* space which holds local data and information about nested procedures within the process. Each text space contains one or more **threads** of execution (a thread is a piece of executable code with access to the data and stack space).

Processes are divided into three types: lightweight, mediumweight and heavyweight. Only the latter two are supported by the RHODOS system.

A mediumweight process shares its text and data space with at least one other process, but its stack is separate from all other processes. The set of processes that share a data space is known as a **team** of Mediumweight Processes. This process type is a compromise between the other two process types. There are several threads of execution, but each has its own stack, and each is seen as a separate process by RHODOS. This gives better inter-thread protection, while still allowing efficient communications.

A heavyweight process may share its text space with other processes, but its data and stack space are separate from all other processes. This is the same as a standard Unix process. There is only one thread of execution in the process, and no part of the address space is shared with another process. Each process has a separate entry in the Process Control Block.

## 5 Semantics of RHODOS' Interprocess Communication

To provide an ability to compare different models of distributed operating system services, the RHODOS interprocess communication primitives should not only be efficient, but they should allow

comparison between different communication schemes. This means that the semantics of RHODOS communication primitives ought to be very general, in order to achieve a flexible basis for constructing different models of process communication.

The RHODOS Distributed Operating System supports both message passing and remote procedure calls. Communicating processes may request different levels of communication reliability and security. Generally, the choices made in RHODOS are [Gerrity et al. 1990]:

- Messages are exchanged between ports.
- Both synchronous (blocking) and asynchronous (nonblocking) **send** is provided. However, the former differs from the usual definition of blocking in that the sender is blocked until the destination communication subsystem receives the message, rather than the destination process.
- Both asynchronous and synchronous **receive** operations are offered.
- A remote procedure **call** (RPC) facility is provided. While it is possible to implement the semantics of an RPC with appropriate **send** and **receive** primitives, providing it as a primitive makes sense from the point of view of ensuring correct usage, and enhancing communication efficiency.
- Primitives offer several levels of message-exchange reliability: **at most once**, **at least once**, and **exactly once**.
- Broadcast and group addressing (multicast) is provided. The group addressing is an addressing at the operating system level.
- A message encryption ability is offered by interprocess communication primitives.
- The protection system operations based on protection models chosen for RHODOS (capability and Access Control Lists), are orthogonal to the communication operations, and do not interact with the communication primitives.

In RHODOS, every active object is a process. Processes communicate only by sending and receiving messages. This is true not only for communication between user processes but also for communication between operating system processes, and between user processes and operating system processes. User processes request operating system services by sending a request message to the appropriate server. The only event in RHODOS is a message arrival [Gerrity 1990]. This implies that a RHODOS' process may only block while awaiting receipt of a message. RHODOS guarantees that a blocked process is unblocked when a message arrives.

Processes create ports in order to receive messages on them. The port always has a system name given by the operating system, although it may have a user name if required. Ports are location-independent and migrate when a process migrates. Each port is a RHODOS operating system object and is protected by a capability or Access Control List, according to the protection system assumed for RHODOS [Vance & Goscinski 1989]. A process may have more than one port and processes may share a port. In some cases the operating system may create a port for a process, which is not accessible to it, in order to supply the mechanism required to block the process which awaiting the completion of a requested quality of service.

The RHODOS port has a wait queue (queue of waiting processes) and a message queue (queue of incoming messages). Only one of these queues can have something in them at a given moment. Thus, if the wait queue has at least one process waiting, then the first arriving message will be received by the longest waiting process (FIFO queue) and it will be dequeued. Alternatively, if a message queue has at least one message, then any process attempting to receive will not wait, but will receive the first message on the queue. A process can wait on at most one port. The operating system synchronizes the queuing operations and guarantees the atomicity of interaction.

The possible quality of service implies different communication semantics. Three types of sending service are currently defined:

- No reliability is requested or guaranteed. The sending process sends a message and does not wait for any kind of acknowledgment. This means that the message may be placed **at most once** in the receiving queue. This quality of service is not available for remote procedure calls.
- The sending process is blocked until either an acknowledgment is received or timeout expires. The acknowledgment indicates that at least one copy of the message has been placed in the receiving port, but does not guarantee that the receiving process has received the message. This quality of service may cause duplicate receiving on the receiver side. The message is received **at least once**.
- As above, the sending process is blocked until either an acknowledgment is received or timeout expires, but the acknowledgment indicates that exactly one copy of the message has been placed in the receiving port (the destination IPC Manager stores each message and its acknowledgment, in order to avoid duplication). The message is received **exactly once** by the receiver.

The encryption parameter shows whether message encryption is requested or not. Encryption is made by the operating system and the user is not able to request his/her own encryption key.

## 5.1 Interprocess Communication Management in RHODOS

The management of interprocess communication in RHODOS is divided into three sections located in the Nucleus and in the Kernel:

- the Nucleus part of the communication subsystem which deals with a port management and message passing of short local messages,
- the IPC Manager which performs remote message passing and local and remote RPC dealing with:
  - \* Address resolution (maps system names to a location)  
The IPC Manager is a part of the distributed location server for ports. The problem of an address resolution is the subject of investigation in RHODOS.
  - \* Group address resolution  
Global group management (creating and updating of groups) is done in the Name Server. A particular address group resolution while sending or receiving is solved by the IPC Manager in the node. Thus, the IPC Manager keeps a partial replication of the Name Server's structure and consistency of the data is ensured.
  - \* Transparency of migration  
The IPC Manager has built-in support for migration. The IPC Manager supports transparent migration by hiding the fact that the source or destination port is being migrated while communication between processes is performed. The IPC Manager also performs fast copying of messages waiting on the port of a migrated process to a new process port in the destination node (as an support for the Migration Manager, which moves the process description to the migration destination). The IPC Manager manages all communication events which can occur between process freezing and port freezing and restores a proper communication status before process resuming on the destination node of migration.
  - \* Server liveness checking  
In RHODOS, the semantics of the `call` operation have been extended by providing server liveness checking. This extension causes a periodical checking of the server to which a `call` has been issued, in order to test whether the server is still alive and the request is still present in the server queue.
- the Network Manager which performs the RRDP transport protocol embedded on top of IP protocol. The Network Manager sends/receives messages with requested quality of service and is responsible for segmentation and resegmentation of messages.



This structure of the communication subsystem is a result of a compromise between the necessary interprocess communication efficiency and the aim of keeping the Nucleus as small as possible. Communication operations which should be very fast (a local exchange of short messages) are placed in the Nucleus but messages with higher qualities of service are sent by the Nucleus to the IPC Manager, which provides proper services. The IPC Manager uses the Network Manager as a subserver. Communication between the Nucleus, the IPC Manager and the Network Manager is consistent with general RHODOS communication model, and is done by local message passing primitives implemented in the Nucleus.

## 5.2 Naming of ports

According to the naming system proposed for RHODOS, each user has his/her own context, created at the user's login. User names of objects have to be unique within a context. The same rule is applied to ports. A port is given a system name when it is created and it is known only by the process which created it. In order to give access to the port for communicating processes, the port has to be known to the Name Server by its user (UName) and system name (SName).

In RHODOS, services are known by ports. The service may be accessible if one of three names are provided:

- system name
- user name
- service attributes

The system name level is useful when a process knows the SName of the server port (e.g. a capability) and may send messages to this port. If not, the process has to get the SName of the server port from the Name Server. This can be done in two different ways: the process may request the port SName giving the port UName or may request the port SName giving the service attributes (e.g. 'print, speed, quality').

There are special Name Server operations available for a process, which sends the port names (UName, SName, and attributes) to the Name Server and make it possible for another processes to get the port SName from the Name Server. Another set of primitives is used to build and destroy communication groups from ports in order to provide a multicast send and to get the SName of this group. Access rights to get the SName (the port SName or the group SName) are set by a separate operation. A process must use a remote procedure call in order to invoke an operation; this means that names can be received in a normal process (client) to process (server) communication.

## 6 Memory Management

A multitasking operating system must perform memory management, so that several processes can share the physical memory, with adequate protection from each other. Hardware is usually used to provide this protection. Also, features such as swapping, paging or virtual memory may be employed to make the system's memory appear larger than it is. These features often depend heavily on the underlying architecture of the machine.

### 6.1 Terms

Unfortunately the terms used in Memory Management discussions are often very vague, so the terms used in the following sections are defined here.

A logical address space (or *space*) is a chunk of memory of arbitrary size, with base address 0. RHODOS subdivides a process into 3 spaces, *text*, *data* and *stack* space, for protection and sharing. Also RHODOS caters for the initialisation of the first *valid* logical address. The process still sees the space starting at 0, but references below the first valid address cause exceptions.

A logical address space is a collection of **segments**. Segments have a maximum size. RHODOS sees segments as machine dependent, and thus may be of fixed size. However RHODOS will handle segments of arbitrary size (up to the maximum) if the machine architecture requires it.

A **page** is a fixed size block of memory, and is very machine-dependent.

## 6.2 RHODOS' Memory Management — Design Decisions

Although techniques such as swapping and paging have been in existence for decades now, their implementation has always been non-portable and sometimes inefficient. For example, the Berkeley Unix operating system uses paging to provide virtual memory; however, it bases paging around the Digital Vax paged architecture, and ports of Unix to other machines have had to preserve this architecture-dependence, even on machines dissimilar to Vaxes. Moreover, Unix performs paging to a fixed-sized disk partition, thereby statically limiting the amount of paging possible.

The RHODOS Memory Management system was designed to overcome these limits. To do this, several decisions were made about memory management [Toomey 1990]:

- **Architecture Independence:** Memory management should be divided into two sections: one to deal with the machine's hardware, and the other to map this onto architecture independent memory objects. The rest of the system should know only about these memory objects. This aids porting of the system to other architectures, and more importantly for a distributed system, allows memory objects to be shared and distributed, even across different architectural platforms.

In RHODOS, the **space** has been chosen as the architecture independent memory object, as it has no limitations. This is mapped onto segments or pages or both by the Memory Manager. This is done by having two or more sections in the Memory Manager, to deal with spaces, segments and/or pages.

- **Efficient Memory Sharing:** If and where the machine architecture supports it, memory management should provide sharing of spaces with architecture-independent types of sharing, such as copy-on-write, read-only, read-write, invalid, and copy-on-read. These protections are mapped as best as possible by the Memory Manager to the protections provided by the hardware.
- **Fast Context Switching:** To perform efficient switching between processes, the nucleus needs a hardware dependent 'memory map' for each process, which is kept in the Process Control Block. This does not mean an exact duplication of the Memory Manager's information, because the nucleus needs it in a form for fast switching, and the Memory Manager does not store it in that form.
- **Internal Modularity:** Each of the levels within the Memory Manager should be independent of the others, so that policy changes within a level do not have a global management effect. For example, the introduction of swapping into the segment level should not affect the space or the page level. Therefore there must be a set of well-defined entry points into each level of the Memory Manager.
- **External Modularity:** Only the Memory Manager has the ability to allocate and manipulate memory. Therefore it too must provide a well-defined set of operations that it can perform on behalf of other processes, such as the Process Manager.
- **Architecture Independent Backing:** Regardless of whether the lower levels of the Memory Manager provide paging or swapping, they do so to an object which is the backing image of a space: the file. This is extremely important for process migration, as there is no guarantee that Migration Managers on both machines use the same form of paging or swapping. Therefore, spaces are migrated, via files on a common file server.

## 6.3 Modularity

RHODOS is designed to be able to take advantage of the types of machine architectures that constitute most of the current computers. This is done by making its memory management code very modular. The code is divided into the following sections: space management, segment management, and page management.

Each management layer provides a standard set of services, which can be accessed by higher levels, through routine calls. For example, the space layer provides such functions as `createspace()`, `delspace()`, `alterspace()`, `linkspace()`, etc. These function may use services from lower layers, or may be able to provide the service 'as is' depending upon the architecture.

Some of the sections may be 'omitted' if the machine architecture does not have the appropriate hardware protection. For example, take a system with segments. A request for a space of size  $X$  arrives at the space section. It asks the segment code to compose this space from one or more segments. If this is possible with the machine's segments, then the request returns successfully, otherwise the segments code returns an error.

For a machine with pages, the segment code is trivial; it passes a segment request directly to the page management code. Again this succeeds if the segment can be built with the pages in the system, or fails otherwise. On a space-based machine, both the segment management and the page management may be omitted.

## 7 Load Balancing in RHODOS

The development of a load balancing facility requires that a distinction be made between policy and mechanisms. A load balancing policy can be described as the problem of deciding *when* to move *which* process *where*. The operation of moving process from one node to another is known as process migration, which is a mechanism for load balancing.

The goals we are pursuing for load balancing in RHODOS are as follows:

- Providing an environment to evaluate a wide range of load balancing algorithms. This implies that the kernel should provide different workload data to support different algorithms.
- Making the replacement of algorithms easy. This means that the interface between the load balancing server and the kernel is a standard one and thus modularity is used to implement the load balancing server.
- Making it possible to tune some parameters, such as the interval of sampling, some thresholds to measure workload, which can be set in experiments by the tester.
- Gathering the statistics about system performance and load balancing costs, which include communication and computation costs of load balancing algorithms. These statistics are used as basic data to analyze various algorithms.

There are a number of problems which should be taken into consideration when designing load balancing facility. The problems are related to *when*, *which*, and *where* [Goscinski 1991].

### 7.1 Load balancing server components

In RHODOS, the load balancing facility contains the following basic components: information gathering, selection and location, and negotiation. These components perform the *when*, *which*, and *where* functions, respectively. Note that a copy of this server can be distributed among all computers involved in load balancing or can be located in one computer if load balancing is performed centrally.

Actually in a distributed approach, the information-gathering and negotiation components conceptually form the interface between the load balancing server and other parts of the distributed system.

The load balancing server is implemented according to some algorithms [Goscinski 1991], [Zhu and Goscinski 1990]. We notice that the tunable parameters for different algorithms vary considerably. Their common parts are time interval for information collection, and time interval for information exchange and thresholds, such as the number of processes for lightly loaded and overloaded computers. In order to investigate the influence of these parameters, a set of primitives have been introduced in RHODOS to assign these parameters.

## 7.2 The load data of the RHODOS load balancing server

The workload data which represents the load and communication pattern in a computer can be divided into four groups; that is, process-, computer-, communication-, and per-process- oriented.

- A *process group* is represented by the number of processes which are ready, running, blocked, along with reasons for their blocking.
- A *computer group* is a set of data which contains the configuration attributes and current available resources of a computer, such as the type of a processor, physical memory space and peripheral devices, free memory, free port, CPU load, etc.; they reflect the computer capacity.
- A *communication group* consists of two parts: local communication and remote communication for each process. We count the number of messages and average size of these message for each node in a time interval. This data is used to find out the dependency among the nodes.
- A *per-process group* reflects process characteristics, which include Tcpu, Twait, and the number of messages exchanged.

In general, every load balancing algorithm needs queue length (process, I/O and message), and CPU load information. The *process* and *computer* parts can indicate the load of a computer. These parameters are usually exchanged among computers. The *per-process* information is mainly used by the selection component to select a process or team<sup>1</sup> to move and the *communication* and *computer* data are used by the negotiation component to assess its local resources whether is able to accept a process or emigrate a process. Note that different algorithms choose different parameters to indicate load. When making load balancing decisions, *when*, *which*, and *where*, it is also necessary to take into consideration the costs of relevant operations. The overheads caused by the load balancing facilities consist of two parts:

- Computation costs, which reflect overhead caused by running load balancing facilities.
- Communication costs, which depend on the costs of gathering data, the costs of exchange of workload, negotiation costs, and migration costs.

A good load balancing algorithm will perform well while limiting the overheads to the lowest possible level. In RHODOS, all of the overheads will be taken into consideration when making a comparison of the different algorithms.

## 7.3 The Load Balancing Server — Migration Manager Interface

The Load Balancing server is responsible for making decisions on *when*, *which* and *where* to migrate the process.

This decision can be the result of either the completion of

- local computations by a load balancing algorithm, or
- the above plus negotiation

<sup>1</sup> A team is a collection of processes which share common resources, usually memory spaces and/or communication ports.

In the latter case, the load balancing server on the destination asks its local migration manager to prepare the execution environment for an incoming process.

It seems obvious that the Load Balancing server should inform its Migration Manager about the action to be taken by the following calls:

- `migrate_out(which, where)`
- `migrate_in(which, from)`

In RHODOS, we have two similar system calls for load balancing server to invoke a migration service.

## 7.4 The Process Migration Manager

In RHODOS, the process migration manager is a facility which implements a mechanism for load balancing. This implies that after the load balancing decisions - when to move which process to where, is made - the selected process must be migrated to the destination computer. Process migration is associated with saving the state information of a process on the source computer, transferring this state to the destination computer, restoring it on that computer, and then resuming its execution in the new environment. This implies that the process migration facilities on both sending and receiving sides must cooperate to perform these functions.

In order to reduce the overhead caused by migration, such as data copying, the kernel processes closely cooperate to minimize the amount of message passing and data copying. We postpone the freezing of incoming communication until the migrating process' environment has been re-established in the destination, and combine communication freezing with the migration of pending messages operations. All of the dangling messages on the source computer are moved to the destination without extra data copying.

Based on the above design strategy, the process Migration Manager on the source computer performs the following steps in migration:

1. freeze a process, i.e., detach a process from its computation environment,
2. copy the state of the process (i.e., pcb, port(s) and spaces) to the destination computer, and
3. when destination rebuilds the above structures and informing source computer of this, ask IPC Manager to freeze ports and copy messages,
4. remove dependency after process migration and redirect the communication.

On the other hand, the process Migration Manager on the destination computer after receiving an invocation from the Load Balancing Server or a message containing the state of the migrating process, should

1. create a process computation environment (by getting new pcb, ports and spaces),
2. inform the source that it is ready to receive messages,
3. on behalf of migration manager, IPC Manager receives the messages from the frozen port of the source computer to the port on the destination computer, and
4. resume the migrating process.



## 8 The File Server

Initially, the File Server will be implemented as a standard process on one of the Unix machines on the network; this is to provide a simple means of file access when developing RHODOS, and to aid in initial debugging. Communication between the RHODOS machines and the File Server is via IP, with RRDP superimposed. RPCs between the RHODOS client and the server conform to a RHODOS File Server RPC protocol, and use **SNames** for all operations. The internal structure of the File Server will be modular, so that in the long term the File Server may become a true RHODOS process, running on a diskful workstation.

The File Server consists of:

- Code to deal with the RHODOS File Server RPC protocol.
- Code to manipulate RHODOS files. This is RHODOS file-specific, so that this section and the previous can be used under RHODOS and Unix.
- RHODOS file → Unix file conversions.
- Unix file manipulation. These two sections mask the Unix dependencies from the first two sections.
- RRDP code.
- Miscellaneous Unix code to deal with the interface to the IP layer in Unix.

## 9 Conclusion

This paper summarises the detailed design of the basic components of the RHODOS distributed operating system, and forms the basis for our implementation, and for the logical and detailed design of the Name, File and Authentication servers.

In summary we can state that the main goal and design assumptions for RHODOS have been made based on the study of a number of existing experimental distributed systems [Almes et al. 1985, Cheriton 1988, Finkel et al. 1986, Popek and Walker 1985, Tanenbaum and van Renese 1985, Tavenian and Rashid 1987], and presented in [Goscinski 1991]. They are as follows:

- Existing distributed operating systems only cover such areas as IPC, naming and protection. This is not enough to solve other problems found in distributed systems.
- It is very difficult to compare different proposals in the area of IPC, naming, and protection.
- Contradictory results published on load balancing are based on simulation and existing systems, for which a Process Migration facility was added after the completion of the whole system.

This paper describes a partly implemented system. However, though our system was not fully implemented and tested, we think that is worth presentation because it contains ideas and approaches not found in other systems. Moreover, we want to develop a distributed operating system not only concentrating on IPC, naming, and file system but also on other design issues which are important to distributed environments.

We believe that the following ideas and solutions are original:

- RHODOS is a test bed as well as a system to develop the final product.
- RHODOS is designed and implemented to study:
  - IPC primitives and their effectiveness in different environments,

- protection systems: access lists, capabilities, mixture of them, as well as nondiscretionary mechanisms,
  - different naming distribution and resolution concepts,
  - load balancing algorithms (based on negotiation and without negotiation).
  - security (communication and authentication).
- The kernel is reasonably small; it is built from a number of servers (managers) and a nucleus to hide hardware (interrupts, traps) and to provide fast interprocess communication (local) and context switching.
  - The kernel implements mechanisms such as process management, memory management, RPC and remote IPC, and process migration (which is a mechanism for load balancing which implements policy).
  - There are three levels of names: user level, system level, addresses (location oriented). User level naming allows the study of distribution and resolution of names; user names are attributed. Syntactic features improve address resolution performance. System names allow the use of capabilities, access lists, or a mixture of them.
  - To achieve good process migration performance to support load balancing, many IPC manager features and issues are oriented toward providing efficient, fast and transparent migration. Processes and teams of processes can be migrated.
  - The Memory manager and other parts are designed and implemented to be moved to other hardware, based on different memory organization.

Finally, we can say that we have learned the following from our work:

- The provision of a load balancing facility requires a very efficient and effective process migration facility.
- The process migration facility should be treated as intrinsic part of a kernel – it cannot be designed and developed after completion of a kernel. It cooperates very closely with IPC manager, memory manager and the nucleus.
- An effective naming system should be developed based on attributed names, to provide yellow and white page services.
- Clear division between user names and system names allows the study of different distribution structures.
- Careful design allows easy study of different protection schemes.
- Load balancing for a “local” distributed system should be based on the distributed approach. Allocation of other resources can be performed based on the centralized approach.

## 10 Acknowledgments

The authors would like to thank Christopher Vance and Chris Lokan for their time and discussions on aspects of this paper.

## 11 References

- [Almes et al. 1985] G. T. Almes, A. P. Black, E. D. Lazowska, J. D. Noe. *The Eden System: A Technical Review*. IEEE Transactions on Software Engineering, SE-11, 43-59.
- [Cheriton 1988] D. R. Cheriton. *The V Distributed System*. Communication of the ACM, 31, (3), 314-33
- [Finkel et al. 1986] R. Finkel et al. *The Charlotte Distributed Operating System*. Computer Science Technical Report #653, University of Wisconsin-Madison, Computer Science Department, 1986.
- [Gerrity 1990] G. W. Gerrity. *A Process Model for RHODOS*. Technical Report CS 90/7, Department of Computer Science, University College, University of New South Wales, Canberra. March 1990.
- [Gerrity et al.1988] G. Gerrity, A. Goscinski, C.J.S. Vance and B. Williams. *The Design of RODOS: A Research Oriented Distributed Operating System*. Technical Report CS 88/17, Department of Computer Science, University College, University of New South Wales, Canberra, September 1988.
- [Gerrity et al.1990a] G. Gerrity, A. Goscinski, J. Indulska, W. Toomey and W.Zhu. *The RHODOS Distributed Operating System*. Technical Report CS 90/4, Department of Computer Science, University College, University of New South Wales, Canberra, February 1990.
- [Gerrity et al.1990b] G. Gerrity, A. Goscinski, J. Indulska and W. Toomey. *Interprocess Communication in RHODOS*. Technical Report CS 90/6, Department of Computer Science, University College, University of New South Wales, Canberra, March 1990.
- [Goscinski 1988] A. Goscinski. *Research and Design Issues of Distributed Operating Systems*. Technical Report CS 88/15, Department of Computer Science, University College, University of New South Wales, Canberra, September 1988.
- [Goscinski 1991] A. Goscinski. *Distributed Operating Systems: The Logical Design*. Addison-Wesley, 1991. In print.
- [Goscinski and Zhu 1990] A. Goscinski and W. Zhu. *The Development and Performance Study of the RHODOS Reliable Datagram Protocol (RRDP)*. Proceedings of the Tenth International Conference on Computer Communication, New Delhi, India, Nov. 4-9, 1990.
- [Popek and Walker 1985] G. Popek, B. J. Walker. *The LOCUS Distributed System Architecture*. Cambridge, Massa: The MIT Press, 1985.
- [Tanenbaum and van Renese 1985] A. S. Tanenbaum, R. van Renese. *Distributed Operating Systems*. Computing Surveys, 17, (4).
- [Tevanian and Rashid 1987] A. Tevanian, R. F. Rashid. *Mach: A Basis for Future UNIX Development*. Carnegie-Mellon University, Department of Computer Science, 1987.
- [Toomey 1990] W. Toomey. *Memory Management in RHODOS*. Technical Report CS 90/19, Department of Computer Science, University College, University of New South Wales, Canberra, May 1990.
- [Vance and Goscinski 1989] C.J.S. Vance and A. Goscinski. *The Logical Design of a Naming Facility for RODOS*. Technical Report CS 89/15, Department of Computer Science, University College, University of New South Wales, Canberra, August 1989.
- [Zhu and Goscinski 1990] W. Zhu and A. Goscinski. *Load Balancing in RHODOS*. Technical

Report CS 90/8, Department of Computer Science, University College, University of New South Wales, Canberra, March 1990.

[Zhu, Goscinski and Gerrity 1990] W. Zhu, A. Goscinski, G. Gerrity. *Process Migration in RHODOS*. Technical Report CS 90/9, Department of Computer Science, University College, University of New South Wales, Canberra, July 1989.

The first part of the paper describes the system architecture and the data flow. The second part describes the implementation of the system. The third part describes the results of the experiments. The fourth part describes the conclusions of the paper.



# Language and Operating System Support for Distributed Programming in Clouds\*

Partha Dasgupta<sup>†</sup> R. Ananthanarayanan<sup>‡</sup>

Sathis Menon<sup>‡</sup> Ajay Mohindra<sup>‡</sup> Mark Pearson<sup>‡</sup>

Raymond Chen<sup>‡</sup>

Christopher Wilkenloh<sup>‡</sup>

<sup>†</sup>Department of Computer Science  
and Engineering  
Arizona State University  
Tempe, AZ 85287-5406  
*partha@enuzha.eas.asu.edu*

<sup>‡</sup>Distributed Systems Laboratory  
College of Computing  
Georgia Institute of Technology  
Atlanta, GA 30332-0280  
*clouds-project@helios.cc.gatech.edu*

## Abstract

The CLOUDS operating system supports a system environment consisting of compute servers, data servers and user workstations. The resulting integrated environment logically simulates a large centralized computing system. CLOUDS supports powerful programming systems that exploit the operating system's persistent memory mechanisms and its transparent nature of distribution.

This paper discusses programming techniques that utilize implicit distribution and implementation details of the major building blocks of the programming systems supported by CLOUDS. In addition, we present system performance measurements and demonstrate the novelty and usability of CLOUDS as a distributed programming platform.

## 1 Introduction

One year ago, in this forum, we published details on the design and implementation of the CLOUDS operating system [WRM<sup>+</sup>89]. In 1989, we were not in a position to adequately present several important issues. They are:

- Why is CLOUDS worthy of consideration as a programming system? What is novel about the programming paradigm?
- How is a system like CLOUDS used effectively?
- What is the performance like?

---

\*This work was supported in part by NSF contract CCR-86-19886.

Although the system prototype was operational, we were not able to fully discuss these issues because the applications that had been built on top of the system were small test applications geared to exercise only a small subset of the operating system's mechanisms. In addition, several key operating system mechanisms had not been fully implemented and the performance of the mechanisms that had been implemented had not been adequately measured.

Since then, most of these missing and incomplete mechanisms have been completed. A user interface has been added, and users are able to run applications on CLOUDS. We now support three different programming environments (one has been completed, the other two are in various stages of implementation). Application programs have been implemented, and they demonstrate the usability of the system.

In addition, we have developed a novel paradigm of writing distributed applications that effectively utilizes the CLOUDS model. This considerably simplifies developing distributed applications, compared to other distributed systems.

## 1.1 Objectives

In this paper, we present the new features of CLOUDS which support a distributed programming paradigm based on large-grained, persistent objects. To this end we:

- Provide details on the CLOUDS C++ (CC++) programming environment (Section 3). This will demonstrate the usability of CLOUDS.
- Show how CLOUDS distributed programs are structured and how CLOUDS' "transparent" distribution facilities are made available to the user (Section 4). In our paradigm, distributed applications can be written in a centralized fashion and yet exploit the parallelism provided by distribution at runtime. This will demonstrate the novelty of the CLOUDS programming paradigm.
- Provide implementation details on the system environment. These details focus on the facilities that have not been previously presented (Section 5 and 6) and provide a flavor of the operating system facilities needed to support distributed programming environments.

In addition, we present performance measurements that demonstrate the feasibility of our approach (Section 7).

## 2 The CLOUDS System Environment

### 2.1 The CLOUDS Paradigm

The CLOUDS distributed operating system is a general purpose operating system that runs on multiple computers connected by an Ethernet. CLOUDS is built on top of a kernel called RA [BAHK87]. RA runs in native mode on Sun-3/60 machines.

CLOUDS is an unconventional operating system that supports an unconventional programming environment. Instead of commonly used artifacts such as virtual memory, files, networking and processes, CLOUDS supports persistent *objects* and distributed *threads*. In addition, CLOUDS provides concurrent executions in shared address spaces, even if the computations execute on separate hosts. CLOUDS also provides distributed synchronization and consistency support.

### 2.1.1 Objects, Threads and Invocations

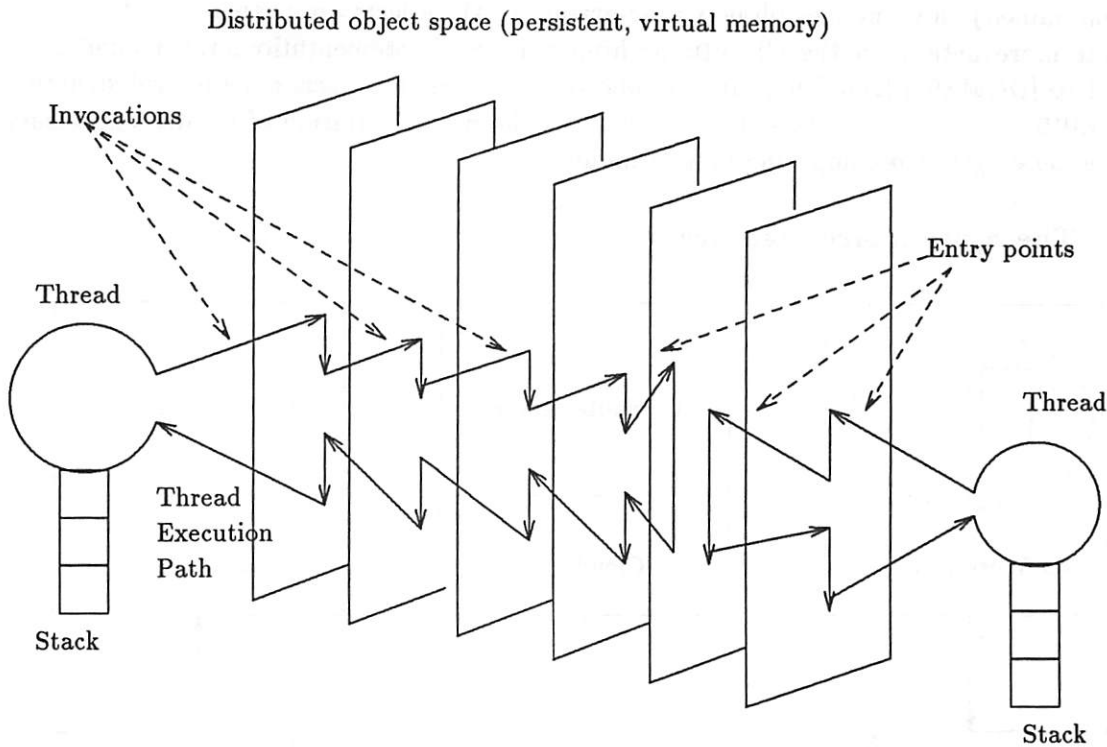


Figure 1: Objects, Threads and Invocations

The CLOUDS programming environment consists of objects and threads (see Figure 1). Each object is a virtual address space that is persistent and visible from all machines. Any number of threads can execute in one object and each thread can access any number of objects through invocations. CLOUDS separates the notion of address spaces (objects) from computations (threads). Objects define the computation and threads perform the execution. Hence, an application developer programs objects not threads.

The CLOUDS invocation facility allows threads to enter an object and run the code in that object. CLOUDS supports two basic kinds of invocations:

- *Local Invocation* causes the target object to be invoked at the same compute server<sup>1</sup> as the thread making the invocation.
- *Remote Invocation* causes the target object to be invoked at some other compute server. The target compute server may be provided explicitly as an argument to the invocation request, or can be implicitly assigned by the system.

In addition, an invocation can be *synchronous* or *asynchronous*. In a synchronous invocation, the calling thread returns to the calling object only after the execution has terminated in the called operation. In an asynchronous invocation, the caller executes in parallel with the callee. Note that the remote asynchronous invocation is a simple but powerful mechanism that can be used to start up distributed computations.

<sup>1</sup>A compute server is a machine that executes CLOUDS threads.

Objects, threads and invocations are used to build distributed object-oriented programming systems on top of CLOUDS. These programming systems support both CLOUDS objects (large-grained) and language objects (fine-grained). All objects are persistent.

For more details on the CLOUDS architecture and implementation, the reader is referred to [DCM<sup>+</sup>90] [DJAR91]. In the following sections, we discuss the logical structure of CLOUDS and show that this structure enables the implementation of facilities necessary for the object/thread computing system model.

## 2.2 The System Architecture

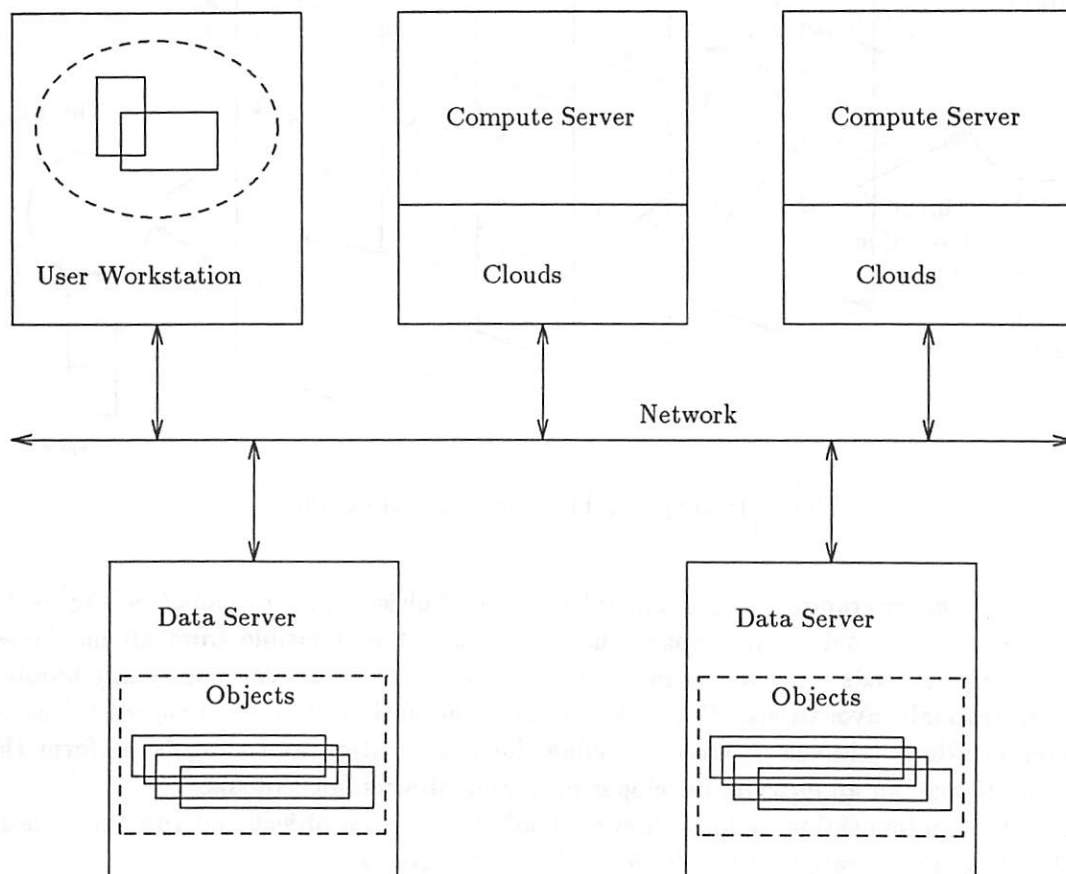


Figure 2: CLOUDS Logical System Architecture

The CLOUDS system integrates a set of homogeneous machines into one seamless environment that behaves like one, large computer. The system configuration is composed of three logical categories of machines, each supporting a different logical function. These machine categories are *compute servers*, *data servers* and *user workstations* (see Figure 2).

The core of the system consists of a set of homogeneous machines of the compute server category. Compute servers do not have any secondary storage. These machines provide an execution service for threads. Secondary storage is provided by data servers. Data servers are used to store CLOUDS objects. The data servers also provide support for distributed synchronization. The third machine category is the user workstation. These machines

provide user access to CLOUDS compute servers.

The logical machine categories do not have to be mapped to physical machines using a one-to-one scheme. Although a diskless machine can function only as a compute server, a machine with a disk can simultaneously be a compute and data server. This enhances computing performance, since data access via local disk is faster than data access over a network. However, in our prototype system, we use a one-to-one mapping, in order to keep the system's implementation and configuration simpler.

## 2.3 DSM and ROI

The compute and data servers interact to provide a distributed operating system environment. These interactions occur through the following CLOUDS operating system mechanisms:

- *Distributed Shared Memory* (DSM)
- *Remote Object Invocation* (ROI)

DSM is used to store and share objects in the system. For example, let us assume that a thread is to be run on a particular compute server. The object in which the thread has to execute must be paged from the data server to the compute server. This requires a remote paging facility, which is provided by DSM. DSM supports the notion of shared memory on a non-shared memory, distributed architecture [AMMR90].

In CLOUDS, there is potential for concurrent invocation of the same object by threads at different compute servers. This results in multiple copies of the same object being used at several compute servers. Hence, DSM has to be cognizant of the need to provide the coherence of shared pages.

The coherence specification of an object  $\mathcal{O}$  being used at two nodes  $\mathcal{A}$  and  $\mathcal{B}$ , requires that  $\mathcal{A}$  and  $\mathcal{B}$  see the same contents of  $\mathcal{O}$ . This is called *one-copy semantics*. The maintenance of one-copy semantics is achieved by coherence protocols that are an integral part of the DSM access strategy [LH86] [RAK89].

Suppose a thread is created on compute server  $\mathcal{A}$  to invoke object  $\mathcal{O}_1$ . The compute server retrieves a header for the object from the appropriate data server<sup>2</sup>, sets up the object space, and starts the execution of the thread in that space. As the thread executes in that object space, the code and data of  $\mathcal{O}_1$ , accessed by the thread, is demand paged from the data server (possibly over the network) to  $\mathcal{A}$ .

If the thread executing in  $\mathcal{O}_1$  generates an invocation to object  $\mathcal{O}_2$ , the system may choose to execute the invocation on either  $\mathcal{A}$  itself or on a different compute server  $\mathcal{B}$ . In the former case, if the required pages of object  $\mathcal{O}_2$  are at other nodes, they have to be brought to node  $\mathcal{A}$  using DSM. Once the object has been brought to  $\mathcal{A}$ , the invocation proceeds. In the latter case, the thread sends an invocation request to  $\mathcal{B}$ , which invokes the object  $\mathcal{O}_2$  and returns the results to the thread at  $\mathcal{A}$ . More details on object sharing is provided in Section 5.3.

The implication of the CLOUDS DSM mechanism is that every object in the system *logically* resides at every node. This powerful concept separates object storage from its

---

<sup>2</sup>The data is retrieved from the data server that contains the object segments. The system-level name of the object contains the identity of the data server.



usage, effectively exploiting the physical nature of distributed systems composed of compute servers and data servers.

The second method of interaction between servers in the system is based on the ROI facility. In order to start a user level computation, a compute server must be selected to execute the thread. The selection is controlled explicitly by the programmer, or implicitly by the system based on scheduling policies. The thread is started on the selected compute server by sending a ROI request to the server. The compute server completes an ROI request by obtaining a copy of the object (via DSM) and executing the thread. In addition, ROI is used by threads to distribute computations by initiating further processing on different system compute servers.

CLOUDS ROI is similar to remote procedure call (RPC) mechanisms supported by other distributed systems such as the V system [Che88]. However, it is more general because a CLOUDS ROI can be sent to any machine and the target does not have to store the called object. This capability is a direct result of using DSM as a repository of objects, as discussed above.

To summarize:

- The DSM coherence protocol ensures that objects are globally accessible and data in an object is seen by concurrent threads in a consistent fashion even if they are executing on different compute servers.
- The ROI facility allows for distribution of computation.

The system structure discussed above allows us to support different kinds of structuring of applications, as described in Section 4. In the next section, we give a brief overview of the specific programming environments supported in CLOUDS.

### 3 The CLOUDS Programming Environments

The programming environment in CLOUDS consists of the CC++, C-Eiffel and CLIDE environments.

#### 3.1 Basic Programming Environment

All three environments support a common object-oriented paradigm. The programmer is provided with two kinds of structuring tools: classes (templates) and instances (objects). CLOUDS objects encapsulate particular application behavior and are large grained. A class is the template that is used to generate instances. Object instances may be invoked by user threads. In order to write application programs for CLOUDS, a programmer specifies one or more CLOUDS classes that define the code and data of the application. The programmer then creates the requisite number of instances of these classes. The application is executed by creating a thread to execute the top-level invocation that runs the application.

A user develops CLOUDS programs using either the CC++ or C-Eiffel language and then compiles them on a user workstation. Once compiled, generated objects are loaded onto a data server, making them available to all compute servers. Any compute node (with initiation from a user) can create instances of these classes. Once a class is instantiated, the resulting object becomes part of the persistent object memory and can be invoked until explicitly deleted.

Threads are started in objects either interactively or by explicit thread creation under program control. A user invokes an object by specifying the object, the entry point and the arguments in a CLOUDS *shell* session running on a user workstation. This CLOUDS shell sends an invocation request to a compute server and the invocation commences. User may communicate with the created thread via an X-terminal window on a user workstation. All output generated by the thread appears on the user terminal window, regardless of where it is executing, and input to the thread is provided through the window (See Section 5.2).

### 3.2 The CC++ Environment

CC++ is a programming language and environment that provides support for CLOUDS classes, objects, instantiation, inheritance and naming [Ana91]. CC++ is an extension of the C++ language [Str86]. To give the reader a flavor of programming CLOUDS objects in CC++, we present a simple example.

In this example, we program a CLOUDS class called `rectangle` represented by using the state variables `length` and `width`. The object has two entry points, one for setting the size of the rectangle and the other for computing its area. The class `rectangle` is defined as follows:

```
clouds_class rectangle
    int    length, width;           // persistent data for rect.
    entry  size (int length, width); // set size of rect.
    entry  int area ();             // return area of rect.
end_class
```

Once the class is compiled, instances may be created. Suppose the `rectangle` class is instantiated, and the instance is called `Rect01`. Now `Rect01.size` can be used to set the size and `Rect01.area` can be called to return the area of this rectangle. The entry point in the object may be called by a command in CLOUDS shell command interpreter. Entry points may also be invoked in a user program, allowing one object to call another.

Objects have user names which are assigned by the programmer when objects are created (compiled or instantiated). In addition, each object has a system-level name (or *sysname*) assigned it by CLOUDS when it is created. Sysnames are used by CLOUDS to refer to objects and have to be provided to the system when an object is invoked. The resolution of user names to sysnames is performed by the CC++ run-time system, via a name server.

CLOUDS objects are referenced by other programs through variables of a special type defined by the language. The variable type is a class instance reference and is called the *clouds object reference class*. It is represented by the suffix `_ref` appended to the CLOUDS class. For instance, the class that refers to an object of class `foo` is called `foo_ref`.

User level names are *bound* to the sysname of an object before invocations can be performed. This is achieved by the `bind` operation in the reference class. The following code fragment details the steps in gaining access to a CLOUDS object `Rect01` and invoking operations on it:

```
rectangle_ref rect;           // rect is a local program handle that refers
                               // to an object of type rectangle
rect.bind("Rect01")           // call to name server, binds sysname to Rect01
```

```
rect.size(5, 10)          // invocation of Rect01
printf("%d", rect.area()); // will print 50
```

The execution of `rect.size` and `rect.area` results in the processing of a local synchronous invocation to the object instance `Rect01`. These operations can be invoked asynchronously by using the syntax `rect!area` or `rect!size`. Remote object invocations (ROI) can be programmed via the virtual node facility (see Section 4.1).

The above example demonstrates programming one CLOUDS object. Since CC++ is an extension of C++, CLOUDS objects can also contain C++ classes and instances. These C++ language entities stored in the address spaces of CLOUDS objects share the properties of CLOUDS objects: they are persistent and can be accessed concurrently via multiple threads that invoke a particular CLOUDS object. Because an object invocation on a CLOUDS object is at least an order of magnitude more expensive than a simple procedure call, a CLOUDS object is appropriate for use as a module that contains several fine-grained entities.

CC++ also provides a variety of other mechanisms to support object programming. These include static type checking, built-in data types, memory support services, user I/O support and facilities to define user interfaces. Some of these facilities are outlined in later sections.

User objects and their entry points are typed by the language definition. Static type checking is performed on the object and entry point at compile time. No runtime type checking is done by CLOUDS.

### 3.3 The C-Eiffel Environment

In addition to CC++, we have implemented a prototype of an extension of the language Eiffel called C-Eiffel [GL90]. Like CC++, C-Eiffel supports large and small grained objects, synchronous and asynchronous invocations, naming and synchronization. While CC++ is primarily used by system program developers, C-Eiffel is targeted for high-level application developers.

### 3.4 The CLiDE Environment

CLiDE (CLOUDS Lisp Distributed Environment) is a distributed, persistent, object-based, symbolic programming environment built on top of CLOUDS [PD90]. CLOUDS object invocation and distributed shared memory mechanisms provide a substrate which is used to build a symbolic processing system that supports cooperation among multiple users and seamless sharing of distinct symbolic processing environments. In addition, CLiDE supports user environment persistence, mechanisms for maintenance of environmental consistency and security, and enables parallel execution within distinct environments.

CLiDE is a set of Lisp based environments. Each environment is implemented as an instance of the CLiDE class. The CLiDE class is composed of a Lisp reader, a Lisp Interpreter and a set of Lisp symbols. Since each object is persistent, the environments retain their state. In addition, CLOUDS ROI is used to provide inter-environment sharing, remote evaluations, concurrency and blackboard communications.

Access to specific CLiDE environment functions and symbols is explicitly controlled, which enables specification of Lisp-based CLOUDS objects with object-like interfaces similar to CLOUDS CC++ and C-Eiffel objects. The ability of CLOUDS objects to interact with

other CLOUDS object types provides a framework for fault-tolerant decision support system construction. CLiDE is currently in the design and prototyping phase and a more complete implementation is expected in 3 months.

## 4 Distributed Programming in the CLOUDS System

While CLOUDS programming environments provide a means to specify application programs, the structure of these applications can vary widely. This section discusses how programmers can organize an application.

An application program, consisting of a set of classes and objects, can be structured in three different ways:

- Treat the CLOUDS system as one integrated, centralized system. Each application is programmed as one or more CLOUDS classes, each with one or more instances. Existing classes can be reused. Each instance is an object that can contain a complete C++ or Eiffel object oriented environment. This is *centralized* programming.
- The programmer can also decide to use the CLOUDS system as a distributed system in which each object is a pseudo-node. Computations are executed in as many nodes as there are objects. This is *explicit* distributed programming.
- The third alternative is to structure the application as one (or more) object(s) as in the centralized scheme, but execute the computation in a distributed manner by starting the computation at several nodes, regardless of where the objects are located. This is called *implicit* distributed programming.

In addition to the above, the programmer can exploit the persistent nature of the objects as well as utilize the concurrency within each object. Centralized programming is the same as traditional sequential object oriented programming and will not be dealt with here. We also do not present persistent and concurrent programming paradigms in this paper. In the rest of this section, we discuss how explicit and implicit distributed programming can be achieved.

### 4.1 Virtual Nodes

The CLOUDS *virtual node* facility is designed to let users target computations to particular virtual nodes. The system is treated as a set of virtual nodes, each having a node number within a sequential range of integers. The programmer, while coding CLOUDS objects, is unaware of the actual physical configuration of the system. Programs request the desired number of nodes from the run-time system associated with the virtual nodes facility. If the system can satisfy this request, it returns the actual number of virtual nodes available to the programmer. The program then partitions its computation based on the number of nodes granted.

To run an invocation on a particular node, the user provides the invocation request with a virtual node number. For example, to synchronously invoke the operation `op` on object `0` on a virtual node identified by `node_num`:

```
0.op (params) at node_num;
```

Similarly, an asynchronous invocation to `op` on object `0`:

```
0!op (params) at node_num;
```

## 4.2 Explicit Distributed Programming

The CLOUDS system can be used as a traditional distributed programming system. For example, consider distributed sorting. One possible algorithm creates  $n$  sorter objects, one on each virtual node. Then, the data is partitioned into  $n$  parts and sent to each of the sorter objects which sort the data and return the results to the main computation. The main computation then merges the  $n$  sorted pieces. This is what we call explicit distributed programming.

Programming an arbitrary algorithm in this fashion is very similar to programming clients and servers in a distributed message system. It involves explicit programming of the distribution and protocol definition to be used for client-server communications and intricate algorithm development. The degree of distribution is also statically defined by the program.

## 4.3 Implicit Distributed Programming

In CLOUDS, using DSM and the different types of invocations<sup>3</sup> it is possible to program distributed applications without using the client server model. This allows distribution to be expressed implicitly, and provides the ability to make decisions on the degree of distribution at runtime.

In implicit programming, the application is structured as one centralized application, typically using *one* CLOUDS object. Implicit distributed programming structures the program as a concurrent program and not a distributed one. Each thread in the concurrent execution runs on a different virtual node, but uses the same object(s). Since DSM provides one-copy coherent memory across machines, the computation will actually work like a concurrent program. However, if the concurrent threads do not heavily share the same pages of memory, the performance will be similar to an explicitly distributed program.

We present a distributed sorting algorithm based on the above idea. The following program implements *one* object called `sorter`, that contains an integer array, which is to be sorted. The object has an operation to sort the array (the entry point `sort`) which is invoked the data needs to be sorted. The code implementing the `sort` operation partitions the computation using virtual nodes.

The operation `subsort` is another entry point in `sorter` that sorts part of the array based on parameters which specify starting and ending points in the array. `subsort` is executed at different virtual nodes depending on the number of virtual nodes available at runtime. When all the `subsorts` terminate, the array is merged.

```
clouds_class sorter
    entry sort(); // sort the entire array
private :
    int array[MAX];
    entry subsort(int i, int j);
end_class
```

---

<sup>3</sup>Synchronous and Asynchronous combined with Local and Remote.



```

sorter::subsort(int i, int j) {
    // sort from array[i] to array[j], in place.
}

sorter::sort() {
    int no_nodes = N;    // number of nodes.
    int seg_size = MAX/N; // can be changed dynamically.
    // Get N nodes from the virtual node facility
    for (int node = 0; node < no_nodes; node++) {
        sorter!subsort(node * seg_size,
                        ((node + 1) * seg_size) - 1) at node;
    }
    //Wait for invocations to terminate; merge sorted segments
}

```

The sub-sorts are concurrently executed using asynchronous invocations. Thus, the sort is executed by multiple threads which execute at a different (logical) compute servers, and perform computation on different parts of the data in parallel. Note that the data itself is encapsulated in a *single* object. The data actually required by each thread migrates to that node automatically, via DSM, as discussed in Section 5.3.

Therefore, programming of this sorter object is achieved without explicit distribution of data, or any knowledge of the actual distribution of the algorithm. Decisions concerning the degree of distribution of the algorithm are made at runtime.

## 5 The Implementation of the System Environment

In our implementation of CLOUDS, compute servers are implemented on top of a minimal kernel called RA and data servers and user workstations are implemented as UNIX processes on UNIX workstations. The compute servers consist of RA and a set of system objects that provide CLOUDS functionality.

This section discusses some of the major building blocks of CLOUDS. These include the invocation handler, the user I/O system and the DSM service routines. In addition, we discuss the distributed synchronization support and user-level utilities that provide compilation support for user objects.

### 5.1 The Invocation System

Objects in CLOUDS are implemented as shared virtual address spaces. Each object has an object header that defines the layout of the object address space. Threads are implemented using local processes. If a thread executes on only one node, then it will be associated with only one process. However, if the thread performs remote object invocations then the thread will have multiple processes executing on behalf of the thread; one on each machine touched by the distributed thread. The *Invocation System* is a system object<sup>4</sup> that handles all the types of invocations.

<sup>4</sup>System Objects are modules that plug into the Ra kernel to build an operating system.

A thread executing in one object invokes another object through a system call. The system call activates the Invocation System, which determines from the system call parameters whether the invocation is to be asynchronous or synchronous, and whether it is a local or remote invocation.

In the case of a synchronous local invocation, the Invocation System saves the state of the current object invocation and brings the object header of the object being invoked into memory. The system uses object headers and page maps to map the new object into the address space of the executing thread. The system then initializes the hardware state (PCB) of the thread, sets up the parameters (if any) and returns control to the executing thread. When the thread resumes execution, it will be executing in the address space of the new object with the program counter set to a location defined by the object header.

Asynchronous local invocations are implemented as a "create-and-invoke" mechanism. The system creates a new thread to perform the object invocation, and the local object invocation is executed using the newly-created thread. Thus, there is no current invocation state to save. The invoking thread returns and resumes execution, executing concurrently with the new thread.

Synchronous remote object invocations are implemented using slave processes on the remote site. The parameters are shipped to the remote site with the invocation request and a slave process is created on that site. The processing on the remote site then continues as if the invocation was a local invocation. Meanwhile, the process on the local site blocks until the remote invocation terminates. When the remote invocation terminates, the return parameters (if any) are extracted and returned to the local site, and the process is unblocked, allowing the thread to resume execution on the local site. Thus, on a synchronous remote invocation, the thread logically crosses the machine boundary to execute on the remote site and then returns to the local site.

In the case of an asynchronous remote invocation, the invocation request and parameters are shipped in the same manner as a synchronous remote invocation. However, the invoking thread does not block and threads execute concurrently.

## 5.2 The I/O Facility

A thread executing in an object has the ability to communicate with the user who started the thread. When a computation is started from a user workstation, a CLOUDS *terminal window* is created on the display under the *X-Windows* system (using a modified *xterm* routine). The CLOUDS I/O facility ensures that all output generated by the thread is displayed on the terminal window (vice versa for input).

I/O statements are contained in object code, since a user programs objects. Threads execute these statements, and are associated with user terminals at runtime. Hence, the same statement when executed by many, possibly concurrent, threads, may produce I/O on different terminals depending upon the terminal associated with each particular thread. User terminal association and the subsequent per-thread I/O direction is handled by the CLOUDS I/O system. The I/O system provides special user objects which are associated with these terminals. Invocations on these I/O objects activate I/O system code in the kernel.

The sysnames of the I/O objects are a part of the properties of a thread. This information is transmitted along with the thread when the thread migrates to another site. Since sysnames uniquely identify the controlling terminal window, the I/O of a computation

reaches the correct window regardless of the current location of the computation. In addition, I/O sysnames are copied to the new thread created by an asynchronous invocation. Thus, all threads of a common parent share the terminal window. The I/O facilities allow changing the I/O sysnames bound to a thread, allowing I/O redirection to other windows, or to objects supporting read/write methods.

Since a computation can migrate, the I/O system has to be stateless. For example, if a computation running on compute server  $\mathcal{A}$  reads a line of input, migrates to compute server  $\mathcal{B}$  using a remote invocation, and then tries to read another line, the next line should be available at machine  $\mathcal{B}$ . Such situations prevent *any* state information about the I/O channels of a thread from being kept at any compute servers. We have a stateless implementation of the uiomgr system object that performs I/O on demand.

### 5.3 Paging and Sharing of Object Code and Data

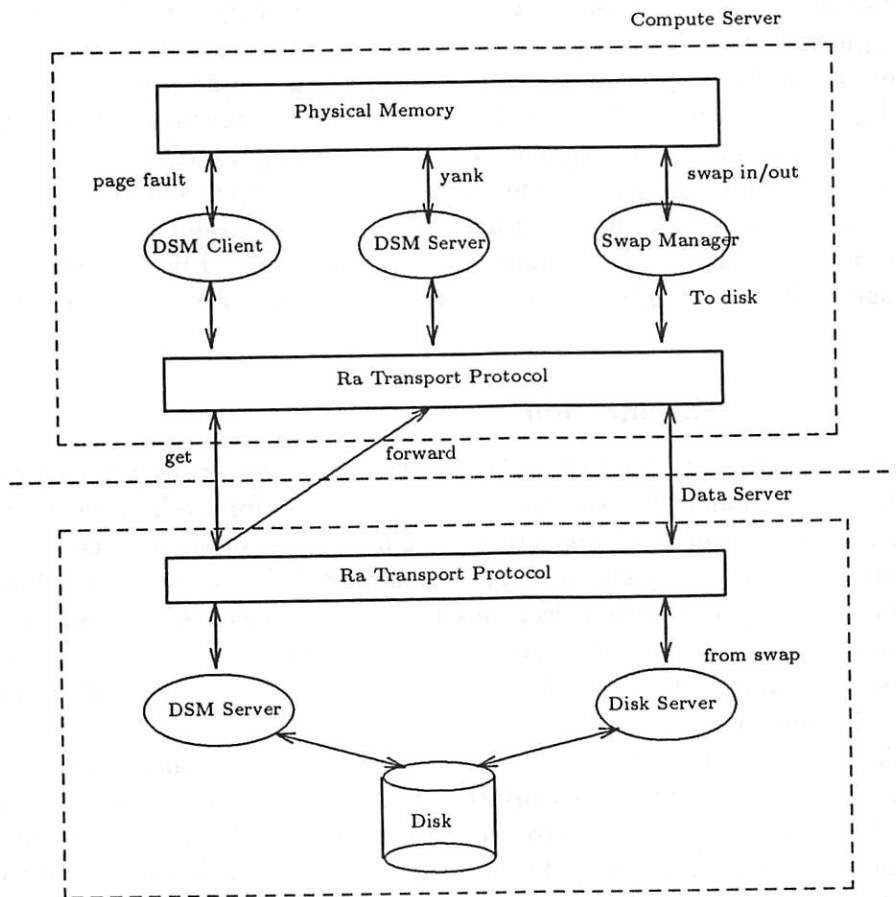


Figure 3: CLOUDS storage system architecture

The DSM system is responsible for making all objects available to all compute servers. It is the software layer between the demand paging system of the RA kernel and the storage daemons running on data servers. The DSM system has several subsystems, namely: *DSM Server*, *DSM Client*, *Swap manager* and *Disk Manager* (see Figure 3). The communications

between the subsystems located on different machines is achieved through the Ra Transport Protocol (RaTP). RaTP is a reliable connectionless protocol that uses message transactions [Wil89].

Each compute server includes a DSM client, a DSM server and a Swap manager as system objects. Each data server runs a DSM server and a Disk Manager as Unix processes.

Suppose a compute server  $\mathcal{A}$  running a computation faults on page  $p$  of data. This fault activates the DSM Client by generating a call to a method in the system object. The DSM Client locates the DSM server containing page  $p$ . The server for any particular page is fixed, systemwide. The site running the DSM server for a particular page is called the *owner* of the page.

Let site  $\mathcal{D}$  be the owner of page  $p$ . The DSM Client on site  $\mathcal{A}$  sends a request to the DSM server on  $\mathcal{D}$ . If  $p$  is currently not being used by any other compute server,  $\mathcal{D}$  sends  $p$  to  $\mathcal{A}$  and the computation progresses. Site  $\mathcal{A}$  now becomes the *keeper* of  $p$ .

At this point, suppose another computation on another site  $\mathcal{B}$  page faults on the same page  $p$ .  $\mathcal{B}$  sends a request to the owner,  $\mathcal{D}$ .  $\mathcal{D}$  forwards the request to the DSM server on  $\mathcal{A}$ , since  $\mathcal{A}$  is the keeper of  $p$ . In response to the forwarded request, the DSM server at  $\mathcal{A}$  unmaps  $p$  from the address space of the thread using the page and sends it directly to  $\mathcal{B}$ . This is called *yanking* the page. If both  $\mathcal{A}$  and  $\mathcal{B}$  use a page concurrently, this page will *shuttle* between  $\mathcal{A}$  and  $\mathcal{B}$  guaranteeing one-copy semantics [LH86] [RAK89].

In the above scheme, each page has one owner (the data server) and at most one keeper (the compute server using it). For read-only pages the constraints are relaxed, and a page can have multiple keepers. Read-write pages can be acquired in read-only mode (via read-mode page faults) allowing better performance when pages are read-shared by several compute servers.

## 5.4 Support for Synchronization

The data space of an object is shared by all computations that execute in the object. Since computations can run in an object concurrently, there is need for mechanisms that provide mutual exclusion and thread synchronization. The data in an object is accessible only by threads executing within the object. Hence, programming of thread synchronization is local to each object. However, the same object may be used by concurrent threads running on different compute servers. Thus, the synchronization must work across machines. This section discusses the implementation of semaphores and locks that provide intra-object, distributed synchronization.

Synchronization support can be provided at the language level using constructs such as semaphores and monitors. The implementation of such constructs, however, needs operating system level support. CLOUDS provides support for synchronization in the form of semaphores and read write locks [Ana]. Each semaphore or lock is identified by the CLOUDS operating system by a name that is composed of two parts: a sysname and an instance identifier. This scheme eases management of these lock names by imposing a logical hierarchy, based on their intended use. The sysname can be the same as the sysname of the object where the semaphore/lock is defined, and each semaphore/lock within the object has an instance identifier. All state information associated with semaphores and read-write locks is maintained by the operating system.

Semaphores support *create*, *P* and *V* operations. Read-write locks support locking in *read* mode or *write* mode, and unlocking. In addition, a *get* operation is provided with

both semaphores and read-write locks. The *get* operation is a directive to cache the state information corresponding to a particular synchronization primitive at the node executing the operation. This operation can be used to improve performance by making use of locality of access to the semaphore or the read-write lock.

## 5.5 From Programs to Objects

In this section, we briefly describe how objects are created from a program specification. In particular, we discuss the implementation of CC++. C-Eiffel is implemented using a similar technique.

CC++ programs are developed on user workstations and are stored as Unix text files. A CC++ program module consists of a class definition file and an implementation file. These programs are converted to C++, using a preprocessor. The converted programs define a CLOUDS class. In addition, the preprocessor generates interface stubs to access this class. These include the CLOUDS object reference class (See Section 3.2) and the information needed to support inheritance of CLOUDS classes. All this information completely defines a CLOUDS class and is stored as part of the environment of the programmer. This environment serves as a library when that CLOUDS class is used or inherited by other CLOUDS classes. C++ programs are compiled with a standard compiler along with the CC++ library which defines, among other things, the CLOUDS system call stubs.

After the compilation of the program(s) to Unix .o files, the programs are linked with the CC++ library using the UNIX link editor (ld). This creates a UNIX executable with the a.out format. The a.out file is then post-processed into segments that adhere to the CLOUDS object format<sup>5</sup>. The program is now stored as two files containing the data segment and the code segment.

The segment files are then loaded on the CLOUDS data server. This is accomplished by adding the segments and the object descriptor (another segment) to the list of segments managed by the data server. At this point, the segments are accessible on the CLOUDS system. Objects represented by these segments can then be invoked or instantiated.

## 6 More Programming Support

In addition to the programming support mentioned in earlier sections, the CLOUDS system supports various types of persistent memory and provides consistency support for persistent objects. These allow CLOUDS programs to use advanced memory structures and define consistency requirements of applications.

### 6.1 Memory Semantics

Persistent memory needs a structured way of specifying attributes such as longevity and accessibility for the language-level objects contained in CLOUDS objects. To this end we provide several types of memory in objects. The sharable, persistent memory is called per-object memory. We also provide per-invocation memory that is not-shared, but is global to the routines in the object and lasts for the length of each invocation. Similarly per-thread memory is global to the routines in the object but specific to a particular thread and

---

<sup>5</sup>An object may contain multiple data segments. The layout and number of segments are under the control of the programmer.



lasts until the thread terminates. This variety of memory structures provides a powerful programming support in the CLOUDS system [DC90].

## 6.2 Consistency Support

The CLOUDS *consistency-preservation* mechanisms present a uniform object-thread abstraction that allows programmers to specify a wide range of atomicity semantics. This scheme performs automatic locking and recovery of persistent data. Locking and recovery are performed at the segment-level and not at the object level. Since segments are user defined, this allows the user to control the granularity of locking. Custom recovery and synchronization are still possible but will not be necessary in many cases.

Instead of mandating customization of synchronization and recovery for applications that do not need strict atomicity, the new scheme supports a variety of *consistency preserving* mechanisms. The threads that execute are of two kinds, namely *s-threads* (or *standard* threads) and *cp-threads* (or *consistency-preserving* threads). The s-threads are not provided with any system-level locking or recovery. The cp-threads, on the other hand, are supported by well defined locking and recovery features provided by the system.

When a cp-thread executes, all segments it reads are read-locked and the segments it updates are write-locked. Locking is handled by the system automatically at runtime. The updated segments are written using a 2-phase commit mechanism when the cp-thread completes. Since s-threads do not automatically acquire locks, nor are they blocked by any system acquired locks, they can freely interleave with other s-threads and cp-threads.

There are two varieties of cp-threads, namely the *gcp-thread* and the *lcp-thread*. The gcp-thread semantics provide global (heavyweight) consistency and the lcp-thread semantics provide local (lightweight) consistency. All threads are s-threads when created. Each operation has a static label that declares the consistency needs of the operation. The labels are S (for standard), LCP (for local consistency preserving) and GCP (for global consistency preserving). The combination of the consistency labels in the same object leads to many interesting, as well as dangerous, thread interactions. The complete discussion of the semantics, behavior and implementation of this scheme is beyond the scope of this paper, and the reader is referred to [CD89].

## 7 Performance

This section presents performance measurements for the CLOUDS distributed operating system. In our environment, compute servers run on *diskless* Sun-3/60 machines; data servers and user workstations are Sun SPARCstation 1 machines running UNIX<sup>6</sup>. Data segment and user I/O access is across an Ethernet, using the RaTP protocol. First, we present the timings for some basic kernel operations. Then, performance measurements for communication and object invocation mechanisms are presented. These timings are compared with corresponding UNIX timings, where appropriate.

Table 1 summarizes the basic kernel operation timings. Context switch time (147  $\mu$ s) involves a context switch between two kernel level processes. Kernel context switching is efficient since the processes exist in the same machine address space. The time to service a

<sup>6</sup>UNIX is a trademark of UNIX System Laboratories, Inc.

Kernel Operation	Time
Context Switch	147 $\mu$ s
Page Fault Service without Zero Fill	629 $\mu$ s
Page Fault Service with Zero Fill	1.5 ms
Null System Call	125 $\mu$ s

Table 1: Basic Timings

page fault when the page is resident on the same node costs 1.5 ms for a zero-filled 8K page and costs 629  $\mu$ s for a non zero-filled page. Such faults do not require network messages.

Communication (CLOUDS)	Time
Ethernet Round Trip	2.38 ms
RaTP Round Trip	4.80 ms
RaTP 8K Page Transfer	11.90 ms
Page Fault service from data server	20.5 ms
User I/O printf	19 ms
Communication (UNIX)	Time
FTP 8K Page Transfer	70 ms
NFS 8K Page Transfer	50 ms
printf over rlogin	31 ms

Table 2: Communication Performance

Table 2 details costs of CLOUDS' communication mechanisms including page transfers and user I/O requests. The Ethernet round-trip time (2.4 ms) involves sending and receiving a short message (72 bytes) between two compute servers. This measurement is taken at the kernel process level, *not* at the Ethernet driver/interrupt level. The RaTP round-trip time of 4.8 ms adds overhead to a simple message but it insures reliability. Timer operations, buffer copying, and context switching account for most of the cost increases. The extra overhead may appear large for a short message, but becomes negligible for larger data transfers. The time to request and receive an 8K page (11.9 ms) demonstrates this point.

The time taken to service a page fault, which requires the page to be fetched from a remote data server, costs 20.5 ms. The page fetch over the network uses the RaTP reliable transport protocol. These costs can be contrasted with the cost of using FTP (reliable) and NFS (unreliable) to transfer an 8K page. The cost of executing a `printf` from a thread and displaying the string on a terminal window is considerably less than Unix `printf` on a connection using `rlogin`. This is because RaTP is a faster protocol, tailored especially for message transactions in the CLOUDS environment.

Table 3 summarizes the costs for local object invocation. Invoking an object for the first time involves bringing the object header and one page of code from the data server to the compute server. Since each page on the Sun-3/60 is 8K, an object invocation involves fetching the object header, the code segment and possibly the data segment. Thus, at least 2 pages are fetched. A null invocation takes about 103 ms., while an invocation that touches a data page takes 130 ms.

The next time the same object is invoked, its pages are cached in memory and invocation time (8.9 ms.) drops sharply. This is because no page fault occurs, which results in no need

Invocation Operations	Time
Synchronous Local Object Invocation	
- 1 <sup>st</sup> time, null	103.9 ms
- 1 <sup>st</sup> time, 1 data page	135.4 ms
- 2 <sup>nd</sup> time	8.9 ms
Asynchronous Local Object Invocation	
- 1 <sup>st</sup> time, return from call	71.4 ms
- 2 <sup>nd</sup> time	17.8 ms

Table 3: Invocation Performance

for network access. Overhead in this case involves switching the address spaces of processes. In general, object invocation costs should be amortized over the lifetime of the object at a particular compute site.

A local asynchronous invocation is measured from the time the invoking thread issues the invocation request to the point the request returns. This involves setting up the object header (paging in one page, on the first invocation) and creating a new thread. The total time of 71 ms. does not involve bringing in code or data pages or waiting for the newly created thread to run. The new thread waits for its time slice before it executes and may wait a long time before it actually executes. Costs for subsequent invocations is less since the object header mapping does not involve network access. However, its cost is larger than a synchronous invocation due to the thread creation overhead.

Remote invocations are almost identical to the local invocations, except that an invocation request is sent to another compute server.

The performance measurements for the CLOUDS distributed operating system show that it is quite competitive with any system that works over a network without local disks. While initial operations are slower, subsequent operations are considerably faster. Thus, the speedup of subsequent operations due to caching provides fast overall execution characteristics when network costs are properly amortized.

## 8 Conclusions

The support for programming distributed objects in a variety of programming languages and environments is one of the strong points of the CLOUDS distributed operating system. The system provides persistent objects that can be used for programming applications. Since the objects are persistent, there is no need for explicitly saving state. In fact, the operating system does not provide for file systems or disk I/O routines available from the user environments.

In addition, CLOUDS distribution mechanisms allow the programmer to implement applications using implicit distribution techniques. Coupled with the orthogonality of compute and data servers, the system design is elegant, easy to use and intuitive. This enhances its usability and represents the novel aspect of the Clouds system environment.

The performance of the system is more than adequate. The compute performance is dependent on the machines used to run the applications; the only bottleneck being the paging of the objects from the data servers. This can be improved with high speed networks or placing the data servers on the same machines as the compute servers. However, keeping

the data servers physically separate has some distinct advantages: orthogonality, uniform access costs and symmetry. We could improve local performance by integrating the compute and data server functions together on one machine. However, this approach would not improve global system performance, since objects located at the combined compute/data server node would still have the same remote access characteristics as they did when the functions were separated. Instead, a solution involving a high-speed network appears more favorable. Thus, in most cases (except if the host is a high-power multiprocessor) the data servers should be kept separate and linked via a high speed network.

## 9 Availability

CLOUDS operating system code for the Sun 3/60 is available via anonymous ftp. For more details, contact our system administrator Mark Reed via email at *clouds-project @ helios.cc.gatech.edu*.

## References

- [AMMR90] R. Ananthanarayanan, Sathis Menon, Ajay Mohindra, and Umakishore Ramachandran. Integrating Distributed Shared Memory with Virtual Memory Management. Technical Report GIT-CC-90/40, Georgia Institute of Technology, 1990.
- [Ana] R. Ananthanarayanan. An Implementation Architecture for Synchronization in a Distributed System. Technical Report (in progress).
- [Ana91] R. Ananthanarayanan. CC++ Reference Manual. Technical Report GIT-CC-91/07, Georgia Institute of Technology, College of Computing, Distributed Systems Laboratory, 1991.
- [BAHK87] J. M. Bernabéu Aubán, Phillip W. Hutto, and M. Yousef A. Khalidi. The Architecture of the Ra Kernel. Technical Report GIT-ICS-87/35, Georgia Institute of Technology, College of Computing, Atlanta, GA, 1987.
- [CD89] Raymond C. Chen and Partha Dasgupta. Linking Consistency with Object/Thread Semantics: An Approach to Robust Computation. In *Proceedings of the 9th International Conference on Distributed Computing Systems*, June 1989.
- [Che88] D.R. Cheriton. The V Distributed System. *Communications of the ACM*, 31(3):314–33, March 1988.
- [DC90] Partha Dasgupta and Raymond C. Chen. Memory Semantics in Large Grained Persistent Objects. In *Proceedings of the 4th International Workshop on Persistent Object Systems (POS)*. Morgan-Kaufmann, September 1990.
- [DCM<sup>+</sup>90] P. Dasgupta, R. C. Chen, S. Menon, M. P. Pearson, R. Ananthanarayanan, U. Ramachandran, M. Ahamad, R. J. LeBlanc, W. F. Appelbe, J. M. Bernabéu-Aubán, P. W. Hutto, M. Y. A. Khalidi, and C. J. Wilkenloh. The Design and

- Implementation of the *Clouds* Distributed Operating System. *Usenix Computing Systems*, 3(1), 1990.
- [DJAR91] P. Dasgupta, Richard J. LeBlanc Jr., Mustaque Ahamad, and Umakishore Ramachandran. The CLOUDS Distributed Operating System. *IEEE Computer*, April 1991. *To appear*.
- [GL90] L. Gunaseelan and R. J. LeBlanc. Distributed Eiffel: A language for programming multi-granular, distributed objects. Georgia Tech Distributed Systems Laboratory, *Submitted for publication*, October 1990.
- [LH86] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. In *Proc. 5th ACM Symp. Principles of Distributed Computing*, pages 229–239. ACM, August 1986.
- [PD90] M. Pearson and P. Dasgupta. CLIDE: A Distributed, Symbolic Programming System based on Large-Grained Persistent Objects. Technical Report GIT-CC-90/62, Georgia Institute of Technology, College of Computing, Atlanta, GA., November 1990.
- [RAK89] Umakishore Ramachandran, Mustaque Ahamad, and M. Yousef A. Khalidi. Coherence of Distributed Shared Memory: Unifying Synchronization and Data Transfer. In *Eighteenth Annual International Conference on Parallel Processing*, August 1989.
- [Str86] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, Reading, MA, 1986.
- [Wil89] Christopher J. Wilkenloh. Design of a Reliable Message Transaction Protocol. Master's thesis, Georgia Institute of Technology, College of Computing, 1989.
- [WRM+89] C. J. Wilkenloh, U. Ramachandran, S. Menon, R. J. LeBlanc, M. Y. A. Khalidi, P. W. Hutto, P. Dasgupta, R. C. Chen, J. M. Bernabeu, W. F. Appelbe, and M. Ahamad. The Clouds Experience: Building an Object-Based Distributed Operating System. In *Proceeding of the Workshop on Experiences with Distributed and Multiprocessor Systems (WEDMS)*, October 1989.



# Experiences with the Liaison Network Multimedia Workstation

Howard P. Katseff, hpk@vax135.att.com  
Robert D. Gaglianella, rdg@vax135.att.com  
Thomas B. London, tbl@vax135.att.com  
Bethany S. Robinson, bsr@vax135.att.com  
Donald B. Swicker, ds@vax135.att.com

*AT&T Bell Laboratories  
Holmdel, NJ 07733*

## *Abstract*

Future computing networks will offer far higher communications performance than current networks provide, making possible a broad range of communications and information retrieval services. Many of these applications will require full-motion video capabilities. The ability to capture these video streams from a digital network has a large effect on the design of workstations. Of particular concern is the ability to allow many overlapping video windows to be simultaneously displayed without flooding the workstation with input.

We make use of an architecture that solves this problem by distributing the workstation's intelligence over a high-performance network to shed computational and communications to other processors on the network. Our initial experience with this architecture using a prototype monochrome workstation on the HPC/VORX local area multicomputer system has been encouraging. The workstation can simultaneously display several windows with 30 frame per second video, each originating from a different processor on the network. Another processor runs the X server, providing text and graphics windows on the workstation by periodically sending images of its windows to the workstation.

## *Motivation*

Broadband ISDN networks<sup>[1]</sup> providing communications at 150 Mbit/sec will become available in the early 1990s and more powerful networks will follow. When combined with large electronic multimedia databases, these networks make possible a broad range of communications and information retrieval services. Applications such as multimedia document browsers<sup>[2]</sup>, shared electronic environments<sup>[3]</sup> and computer-assisted conference systems will become ubiquitous. Computation will be seamlessly distributed throughout the network, causing the workstation to be viewed as a user-friendly means of accessing network resources rather than a vehicle for computation and allowing us to create sophisticated and powerful workstation user interfaces.

All the applications mentioned above are communications-intensive, requiring either continuous video images or fast-changing bitmap images. Some of these applications require the ability to multicast the same video stream to several recipients. Our interests lie in the design of systems and devices that can be used to support these communications-intensive applications. Of fundamental concern is the workstation, the device that presents and obtains information for the

user. As HDTV receivers with compact-disk quality audio become common household items, computer users will demand that the video and audio quality of the workstation keep up with that of their home entertainment systems<sup>[4]</sup>. Display screens will be larger, have higher resolution, and be able to simultaneously display far more colors than those of current workstations. Workstations of the 1990s will be true multimedia workstations, allowing the simultaneous display of windows with full motion video, fast-changing bitmap images, CD-quality audio, graphics and text, all of which may originate from distant sources.

This paper describes a prototype workstation with high communications bandwidth, fast file access, and high-fidelity audio capabilities. This prototype has been used to experiment with new software and hardware implementation strategies as well as with the implementation of next-generation applications. A distinguishing feature of this workstation is its distributed architecture<sup>[5]</sup>. By distributing the workstation's intelligence over a high-performance network to shed computational and communications to other processors on the network, we are able to overcome some of the bandwidth limitations of current networks.

### *Capabilities*

One of the goals of this work is to determine the feasibility of a distributed architecture in which the workstation does no computation, but simply accepts video or other bitmap images from other processors on the network<sup>[5]</sup>. Such an architecture greatly simplifies the workstation and permits it to be constructed in hardware in a timely fashion. It meshes well with the "open architecture receiver" that permits identical hardware to be used for both computer displays and HDTV consumer televisions<sup>[6]</sup>. We make use of distributed processing to implement basic workstation functions and high-level applications.

The major limitation of our experimental testbed is the network access bandwidth. The prototype workstation is based on a Synergy Microsystems PEGC video board with a 1280×1024 pixel frame buffer connected to the local bus of its 33 MHz Motorola 68020-based single board computer. This workstation is connected to a network of processors by the VORX operating system and the HPC network<sup>[7]</sup>. The current HPC/VORX configuration provides communications and distributed processing with 80 Motorola 68020 single board computers and 10 Sun hosts. By using user-defined communications objects, programs on any two processors can send small messages to each other with a latency of under 40 µsec and obtain a throughput of over 3.2 Mbyte/sec for large messages.

Though these bandwidths and latencies are more than sufficient for distributed processing applications<sup>[8]</sup>, they are not good enough for transmission of high-quality color video. Even with eight-bit color or gray scale we would be limited either to video windows of a tiny size or to full-screen windows with a slow update rate. These considerations have forced us to use a monochrome (bi-level black and white) display. Because we can pack each pixel into a single bit, HPC/VORX is fast enough to allow remote processes to refresh nearly the entire screen 30 times per second.

As a demonstration of the capabilities of our prototype workstation we have run the following experiment. The processor with the video board runs a program that reads bitmap data from the HPC and copies the data to its frame buffer. A window system is provided by an X server<sup>[9]</sup> that resides on a different processor and sends real-time bitmap images of its entire screen to the display processor. While the X server is running, a process on a third processor sends images of a digitized video stream (an AT&T television commercial) to one of the windows on the display. The processor controlling the commercial updates its 480×640 pixel window 30 times per second while the X server updates the rest of the 1280×1024 screen 10 times

per second. Error-diffusion dithering allows us to present a surprisingly good rendition of the commercial on the monochrome monitor.

Because this dithering technique cannot be executed in real-time, we have stored the dithered commercial on computer disks, using approximately 80 Mbytes of storage. Unfortunately, a single processor can only read data from its 5¼ inch SCSI hard disk and send it on the network at a rate of 1 Mbyte/sec. We obtain the required rate of 1.3 Mbyte/sec by interleaving the video data on two disks, each located on its own processor. The data is synchronized by a third processor that sends a short synchronizing message each frame time (thirty times a second), alternating between the two processors with the disks. The disk processors start the transmission of their next frame each time this message is received. The processor that controls the video stream for the display receives and merges the two video streams sent by the disk processors.

While the commercial is being displayed, we simultaneously transmit its sound track on the network. We have built hardware that conforms to the SONY/Philips digital interface format and captures the 44.1 kHz digital output of a compact disc player or digital audio recorder and makes it available to a program running on one of our processors. This program copies the data to the HPC network. We have also built hardware that a program can use to produce a data stream that feeds into the digital direct input of an audio amplifier. In our demonstration, the audio data for the television commercial is stored on disk along with the video images and is transmitted in real-time via the HPC to the processor with the audio output hardware. The data rate required for the audio is 200 kbyte/sec, an order of magnitude less than for the video data.

The current prototype is capable of simultaneously displaying multiple video streams in different windows. These video windows can be manipulated as ordinary X windows and may overlap and obscure each other or other X windows. We are also able to use the HPC hardware multicast mechanism to have a video stream displayed on several workstations at the same time. The video windows associated with a stream may appear in a different position on each workstation display and may be clipped differently.

The photograph in Figure 1 shows the workstation display when running the experiment. Each of the five video windows on the workstation is actually cycling through the 1800 frames of the commercial in real-time: once per minute.

### *Implementation*

Since we want our workstation to work with a variety of multimedia applications, its architecture allows many windows to be simultaneously displayed at video rates. It is interesting to note the effect of allowing many overlapping video windows originating from the network, where only a small portion of each window is visible on the display. Some authors have proposed having each video source sent directly to the workstation and for the workstation to perform clipping of overlapping windows<sup>[10]</sup>. In this case, the workstation must read each video image in full from the network and discard the information describing the obscured parts of the windows, thereby requiring the workstation to be capable of processing incoming images at a much faster data rate than the update rate of the display. If completely obscured windows are allowed, the amount of input that the workstation must process becomes unbounded. Our architecture avoids this problem by limiting the amount of data sent to the workstation's display.

We limit the amount of data sent directly to the display by shedding its communications load to other processors in the network. All images are positioned and clipped into windows before

arriving at the workstation. This function is performed by a *vfilter*, a processor obtained from the processor pool that accepts a full-size window from a video source, clips its contents as specified by the window system, and sends the clipped images to the appropriate position on the workstation display. In this way, each pixel on the display is written to by no more than one *vfilter*, effectively limiting the rate of input to the display.

In our experiment, the data for each video window, as well as that for the X server is processed by a *vfilter* before being sent to the display. Each video window may be partially or fully obscured by other video or X windows, and its *vfilter* sends only the visible portion of the image to the display. The *vfilter* for the X server sends only the portion of the screen not obscured by a video window to the display.

Each workstation uses an additional processor, an *area manager*, that communicates with each of the *vfilters* and with the X window system. Whenever the position of a window is changed, the X window system informs the area manager of the change. The area manager recomputes the visibility of each window and sends to each *vfilter* the position and clipping for its window. The area manager runs fast enough that the changes appear to be instantaneous. Figure 2 shows how the video sources, *vfilters*, and the workstation's display are interconnected in this experiment. The solid lines in that figure indicate logical network connections through which video images flow and the dotted lines indicate control information.

This architecture makes it possible to multicast video to several recipients. Each recipient has his or her own workstation, with a *vfilter* for each window. The originator of the video images uses the hardware multicast mechanism to send the full-size image to each of the *vfilters*. Each *vfilter* does the positioning and clipping that is required for its workstation. In this way, each workstation can have its own view of the multicast video stream. The diagram in Figure 3 shows how a video source could be multicast to two workstations.

### *Limitations*

Our prototype workstation has several limitations that must be corrected in the color workstation that we are planning to build. The most annoying problem is that the pointer (the icon that follows the movement of the mouse) disappears when it is moved into a video window. This occurs because the pointer is placed on the screen by the X server, but the X server does not write into the portions of the display occupied by video windows. Several fixes for this problem have been considered. One is to remove the pointer from the control of the X window system and make it a separate window whose visibility is maintained by the area manager. This would require high performance from the area manager. A mouse in continuous motion typically reports its position 24 times a second, so the area manager must compute and send visibility information at this rate. An alternative solution would be for the pointer icon to be managed directly by the workstation's display hardware. The display would ignore data sent to the position of the pointer icon, and would paint the icon there instead.

There is yet another problem with the pointer: its movement is jerky when displayed by the X server. This occurs because X only updates the display ten times per second. While fast enough to display text typed at the keyboard with no perceptible delay, this update rate is not fast enough to display all the intermediate pointer positions when the mouse is moved. This problem will be corrected when we have enough bandwidth to update the X windows 24 times per second.

The other limitations result from the use of a monochrome display. One is that the edges of the video windows and the output of the X server are constrained to begin and end horizontally at a location that is a multiple of 32 pixels from the left edge of the screen. This is done to



simplify and optimize data transfers between the HPC network and the display. If windows were allowed to start at arbitrary bit positions, a bit-shift and mask operation would be required for each word in data transfers. This problem occurs because monochrome images are packed one bit per pixel and will not happen with a color display.

The final problem is that video sources require a large amount of processing prior to display. Our one minute video sequence was originally 1800 frames of color data. This had to be transformed to a gray-scale image and then dithered for display on our monochrome monitor because each pixel on the display is either black or white<sup>[11]</sup>. This dithering is computationally expensive. It required about one week of computer time on a Sun 3/160 processor to process the video sequence. This makes the use of cameras or other real-time video sources impractical with our monochrome workstation.

### *Insights*

A key contribution to the success of our multimedia workstation is the architecture of the HPC network. The HPC implements flow-control entirely in the network hardware. This makes loss of messages due to buffer overflow impossible. Because the HPC also provides reliable message transmission, the need for implementing recovery mechanisms in the communications software is entirely eliminated<sup>[12]</sup>. The VORX operating system provides a general interface for user-defined communications objects. Processes can access the hardware input and output registers from their applications, eliminating the overhead of supervisor calls into the kernel, and can specify interrupt service routines to handle incoming messages.

Because data transmission is reliable, simple and efficient user-defined objects are used to implement the workstation. For instance, the processor with the frame buffer and display runs only an interrupt service routine. Each incoming message contains a line segment of some length and one pixel high for display. The routine uses programmed I/O to read the first words of the message. These contain the coordinates of the leftmost pixel of the segment and its length. The routine then reads the rest of the message directly from the network to the appropriate location in the frame buffer. This scheme involves no unnecessary copying of data and a minimum of overhead. The vfilter and disk processing programs similarly make use of simple and direct communications schemes that have no protocol overhead.

With a 33 MHz Motorola 68020 processor, the display processor can read image data from the network and place it in its frame buffer at a rate of 3.2 Mbyte/sec (25 Mbit/sec). This is about three times faster than the general-purpose communications protocol provided by VORX<sup>[13]</sup>.

The vfilter program does twice as much communications for each message as the frame buffer program because it both reads and writes each line segment. Its throughput has been measured at 1.7 Mbyte/sec. The vfilter program occasionally receives messages from the area manager to change the visibility of its window and performs a computation of several hundred microseconds to update an internal table describing how to process incoming image data. An early implementation had the processor read and discard incoming image data while it was updating its table. This cost about half the processor cycles and doubled the time to update this table. We were able to eliminate this overhead using a feature of the network multicast mechanism that enables a processor to turn off receipt of multicast messages. Because only the video is multicast, the vfilter program is able to have the network filter out the incoming video messages while the vfilter updates its tables.



## *Conclusions*

We have demonstrated the feasibility of transmitting digital full motion video images over a network to a workstation and of integrating them with an existing window system. Though we adopted the extreme view of forcing all images, including the output of the X server, to be shipped across the network to the workstation, our experimental workstation performs reasonably well. This validates our workstation architecture and confirms our techniques and algorithms for distributed management of window visibility.

The video images were stored on ordinary computer disks. We demonstrated the feasibility of interleaving the data across multiple disks on different processors to provide sufficient throughput for full motion video. A simple synchronization method is able to provide video and sound that are synchronized with each other and flow smoothly.

We are experimenting with the transmission, display, and storage of full motion color video on our network. Our eventual goal is to build multimedia color workstations and fast multimedia file servers that are able to deal with large numbers of high-bandwidth data streams.

## REFERENCES

1. Weinstein, S. B., "Telecommunications in the Coming Decade," *IEEE Spectrum*, November 1987, pp. 62-67.
2. Judith, H. I., "Multi-media Information Services," *IEEE Commun.*, June 1988, pp. 27-44.
3. Ensor, J. R., et. al., "The Rapport Multimedia Conferencing System—A Software Overview," *Proc. Second IEEE Conf. on Comput. Workstations*, Washington, March 1988, 52-58.
4. Jobs, S., "Computing in the '90s," talk given at AT&T Bell Laboratories, Murray Hill, NJ, December 1989.
5. Katseff, H. P., et. al., "The Liaison Network Multimedia Workstation," in preparation.
6. Solomon, R. J., "Broadband Communications as a Development Problem," *Proc. Internat. Seminar on Science, Technology and Economic Growth*, Organisation for Economic Co-operation and Development, Paris, June 1989.
7. Gaglianello, R. D., et. al., "HPC/VORX: A Local Area Multicomputer System," *Proc. Ninth Internat. Conf. on Distr. Comput. Sys.*, June 1989, Newport Beach, 542-549.
8. Renesse, R. van, et. al., "Performance of the World's Fastest Distributed Operating System," *Operat. Syst. Rev.* **22**,4, Assoc. Comput. Mach., October 1988, 25-34.
9. Scheifler, R. W., and Gettys, J., "The X Window System," *ACM Trans. on Graphics* **5**,2, April 1986, 79-109.
10. Kamata, H., et. al., "Communications Workstations for B-ISDN: MONSTER (Multimedia Oriented Super Terminal)," *Proc. Global Commun. Conf.*, Dallas, November 1989, 959-964.
11. Judice, C. N., "Dithervision – A Collection of Techniques for Displaying Continuous Tone Still and Animated Pictures on Bilevel Displays," *Sid. Int. Symp. Digest of Tech. Papers*, 1975, 32-33.
12. Katseff, H. P., Gaglianello, R. D., Robinson, B. S., "The Evolution of HPC/VORX," *Proc. Second ACM SIGPLAN Symp. on Princip. and Pract. of Parallel Programming (PPoPP)*, Seattle, March 1990, 60-69.
13. Gaglianello, R. D. and Katseff, H. P., "Communications in Meglos," *Softw. Pract. and Exper.* **16**,10, October 1986, 945-963.

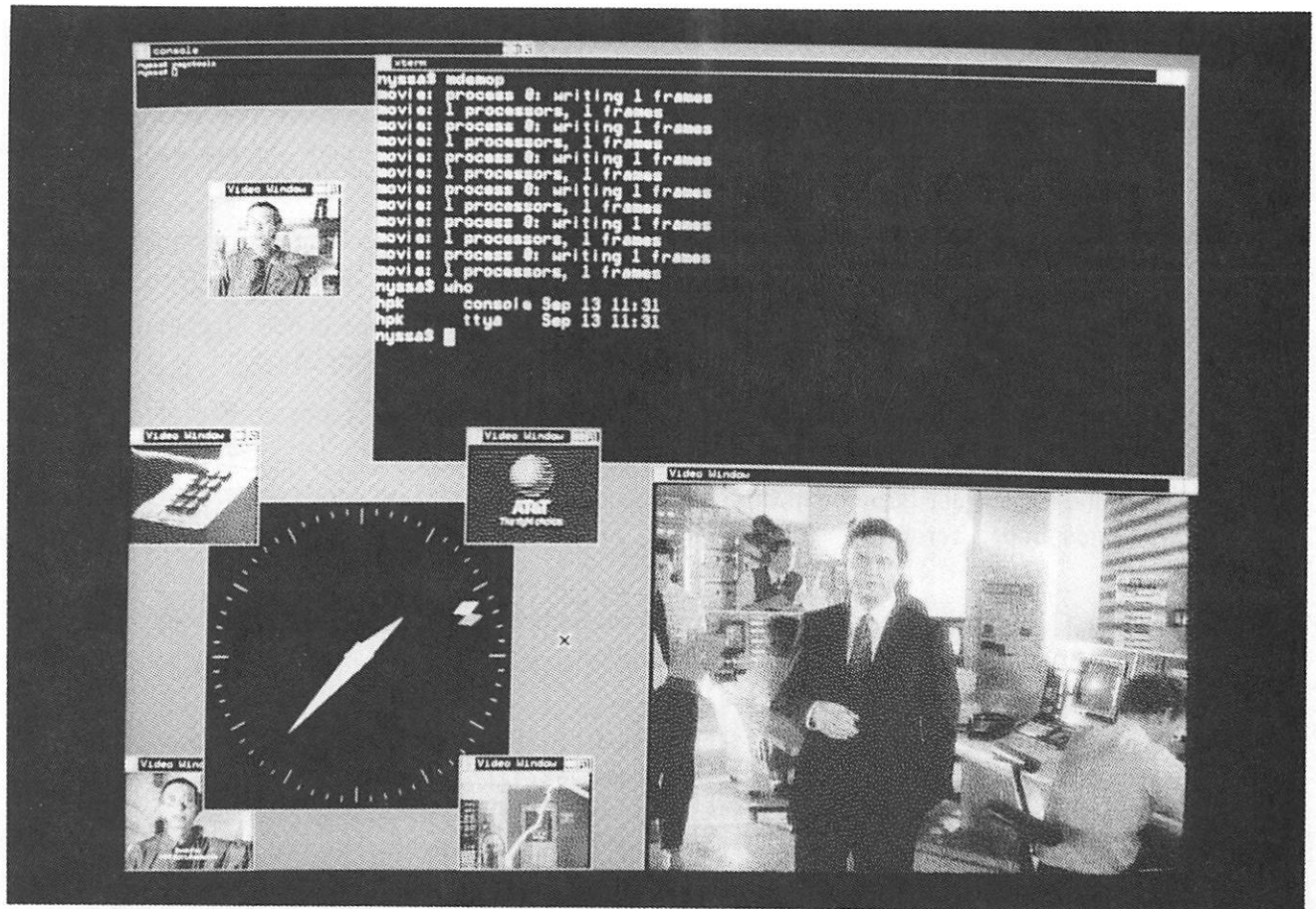


Figure 1. Typical Display from the Liaison Workstation

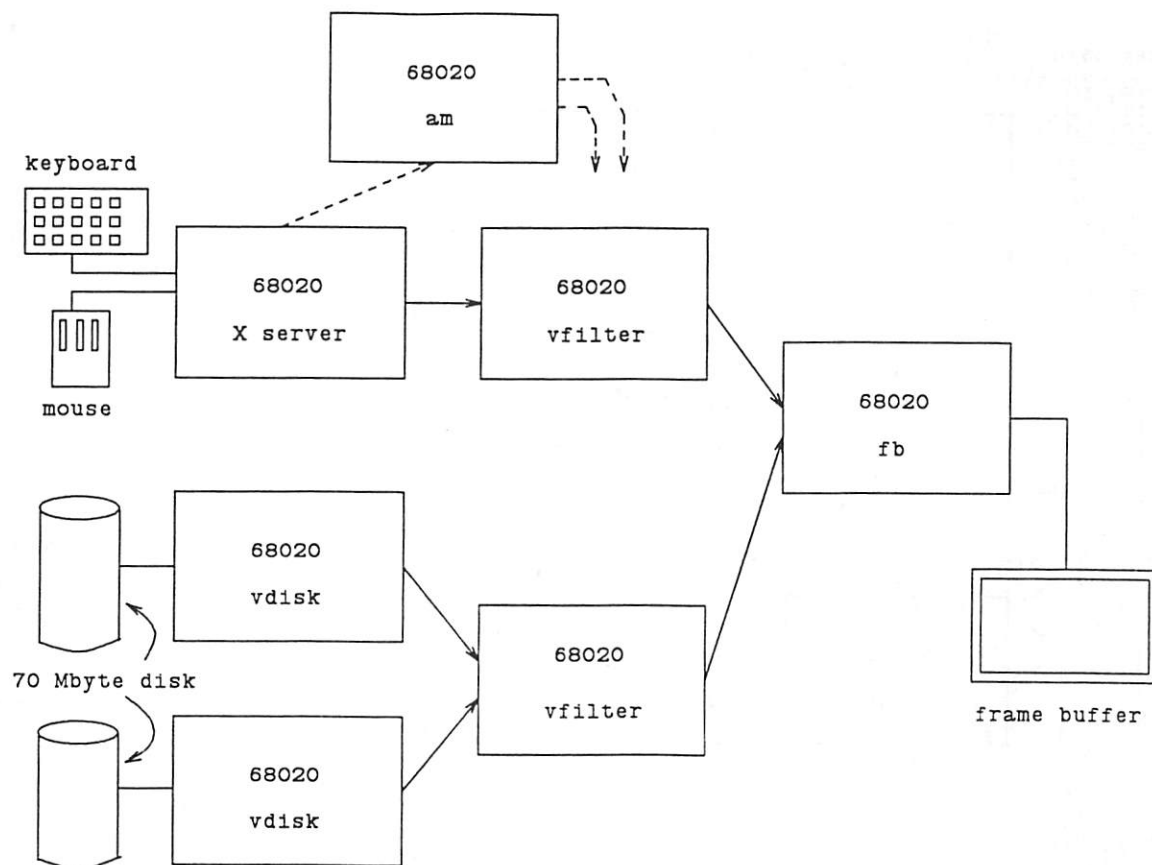


Figure 2. Architecture of the Liaison Workstation

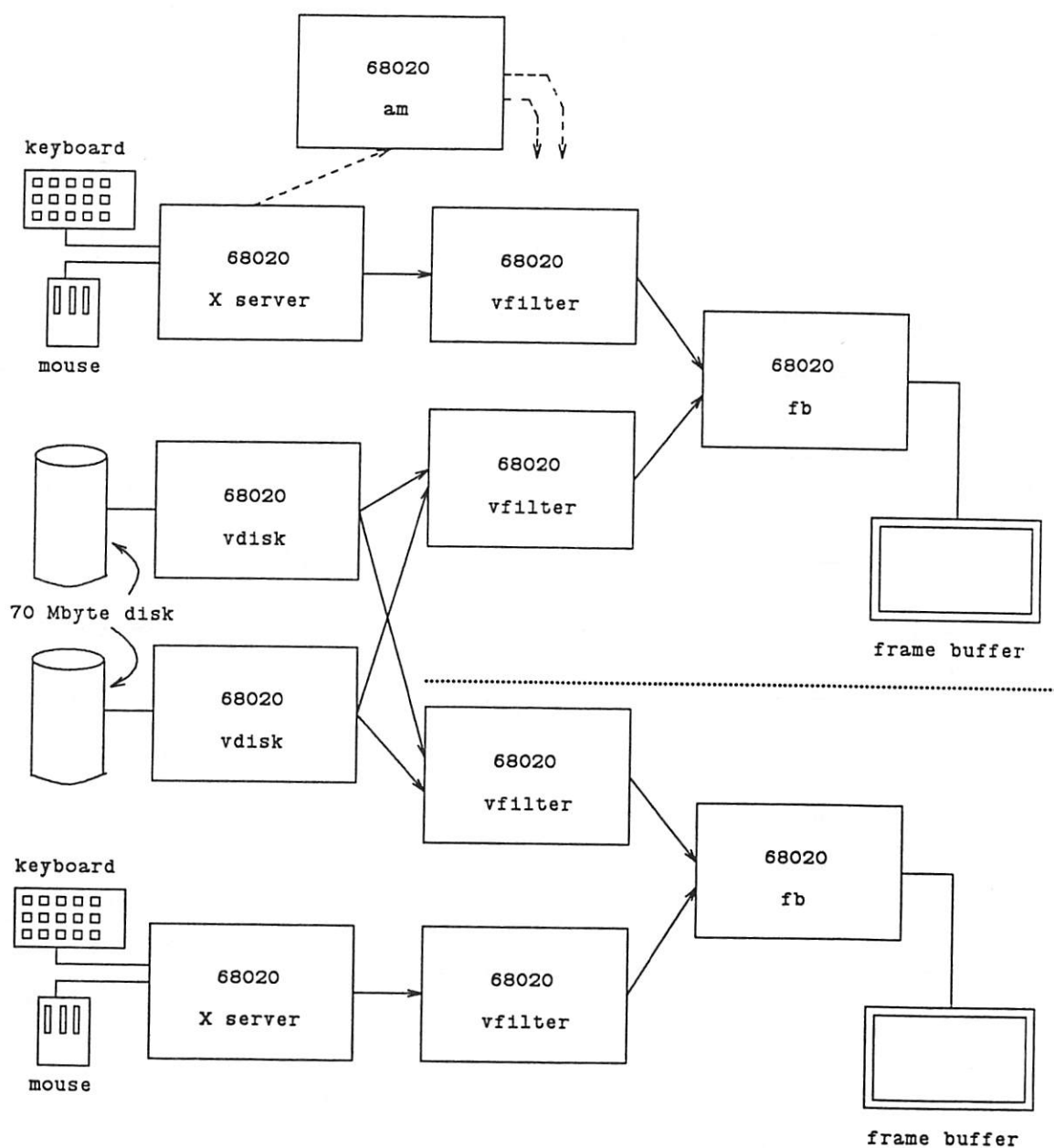


Figure 3. Video Multicast to Two Workstations



## *The USENIX Association*

The USENIX Association is a not-for-profit organization of those interested in UNIX and UNIX-like systems. It is dedicated to fostering and communicating the development of research and technological information and ideas pertaining to advanced computing systems, to the monitoring and encouragement of continuing innovation in advanced computing environments, and to the provision of a forum where technical issues are aired and critical thought exercised so that its members can remain current and vital.

To these ends, the Association conducts large semi-annual technical conferences and sponsors workshops concerned with varied special-interest topics; publishes proceedings of those meetings; publishes a bimonthly newsletter *login*; produces a quarterly technical journal, *Computing Systems*; co-publishes books with The MIT Press; serves as coordinator of an exchange of software; and distributes 4.3BSD manuals and 2.10BSD tapes. The Association also actively participates in and reports on the activities of various ANSI, IEEE and ISO standards efforts.

*Computing Systems*, published quarterly in conjunction with the University of California Press, is a refereed scholarly journal devoted to chronicling the development of advanced computing systems. It uses an aggressive review cycle providing authors with the opportunity to publish new results quickly, usually within six months of submission.

The USENIX Association intends to continue these and other projects, and in addition, the Association will focus new energies on expanding the Association's activities in the areas of outreach to universities and students, improving the technical community's visibility and stature in the computing world, and continuing to improve its conferences and workshops.

The Association was formed in 1975 and incorporated in 1980 to meet the needs of the UNIX users and system maintainers who convened periodically to discuss problems and exchange ideas concerning UNIX. It is governed by a Board of Directors elected biennially.

There are four classes of membership in the Association, differentiated primarily by the dues paid and services provided.

For further information about membership or to order publications, contact:

USENIX Association  
2560 Ninth Street, Suite 215  
Berkeley, CA 94710

Telephone: 415 528-8649  
Email: [office@usenix.org](mailto:office@usenix.org)  
Fax: 415/548-5738

### *USENIX Supporting Members*

Aerospace Corporation  
AT&T Information Systems  
Digital Equipment Corporation  
Frame Technology Corporation  
mt Xinu

Matsushita Graphic Communication Systems, Inc.  
Open Software Foundation  
Quality Micro Systems  
Sun Microsystems, Inc.  
Sybase, Inc.

